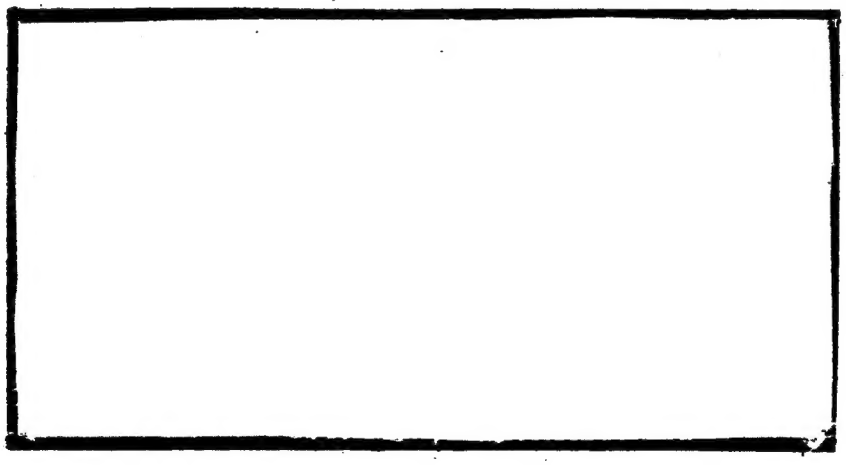

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base

UNCLASSIFIED

AFIT/EN/TR94-07

Air Force Institute of Technology

A Formal Extension to Object Oriented Analysis Using *Z*

Thomas C. Hartrum Paul Bailor

October 7, 1994

Approved for public release; distribution unlimited

A Formal Extension to Object Oriented Analysis Using *Z*

Thomas C. Hartrum & Paul Bailor
Department of Electrical and Computer Engineering
School of Engineering
Air Force Institute of Technology

October 7, 1994

Abstract

This report describes extending an informal object oriented analysis model with formal constructs. Formal methods have evolved over recent years to the point of being supported by specific languages, both mathematical such as *Z* [1] [2] [3] [4] [5], and executable wide-spectrum languages such as Refine [6]. However, the application of such models is still difficult. This is partly due to the abstractness of such specification and the lack of well defined methodologies for applying them to real problems.

One of the appeals of more informal methods is the ability to visualize a system through the use of graphical representations. Also, methodologies for developing these informal models are better established. In particular, the object oriented paradigm has become popular in recent years.

This report documents an integration of the two approaches. Our modeling language is an extension of the informal object oriented model to include formal specification of its basic constructs. Our methodology builds on that already evolving for object oriented modeling. The result is a process that is easy to understand and apply, while resulting in a formal specification.

Contents

List of Figures

List of Tables

1 Introduction

The development of formal specifications for complex software systems offers the potential for making software engineering as mathematically precise as other engineering disciplines are required to be. It also provides an environment in which proof obligations on the specification can be conducted.

Formal specifications more clearly show the benefits of reuse at higher levels of abstraction. Formal specifications tend to be much more loosely coupled than the specifications produced by informal methods. They serve to specify classes of possible behaviors as opposed to very specific behavior which leads to very specific solutions, which means they can be more readily reused. Additionally, their mathematical basis lends them to be more easily combined by mathematical means in order to compose higher level behaviors. This capability is especially important if we are ever to realize the benefits of domain analysis and domain modeling. The formality of the specification languages also offers potential for greatly increased automated capabilities over current generation CASE tools. In general, this is true from both an analysis of properties perspective and a generation/synthesis of lower level code perspective.

Formal methods have evolved over recent years to the point of being supported by specific languages. At one end of the spectrum there exist mathematically-based, non-executable languages such as *Z* [1] [2] [3] [4]. At the other end of the spectrum there exist mathematically-based, wide-spectrum languages such as Refine^{TM} [6] which are also executable. However, learning to use formal methods can often be difficult. Part of the problem is the abstractness of the formal-based languages, a problem that can be overcome in the long run by providing software engineers with a better background in applied discrete mathematics. A more substantial problem to overcome is the lack of well defined methodologies for applying formal methods to real problems. While progress is certainly being made in this area, much still needs to be accomplished.

This paper describes an approach that integrates well-established informal methods with formal methods. In particular, the object-oriented system modeling approach based on the book by Rumbaugh, et.al. [7] is extended by using the *Z* formal specification language to produce a formal-based, object-oriented specification. Our modeling language is an extension of the object-oriented model to include formal specification of its basic constructs, and our methodology builds on that already evolving for object-oriented modeling. The result is a process that is easy to understand and apply, while resulting in a formal specification.

It should be noted that other researchers are also working on defining variations of *Z* to support the object oriented paradigm [8]. Our approach does not extend the basic *Z* language, but uses it to augment the Object Modeling Technique (OMT) as presented by Rumbaugh, et.al. [7]. As described in this report, our emphasis is on the requirements analysis phase of the software development life cycle. However, the form of our model is to some extent driven by our ongoing research in transformation systems to generate executable code from these formalized specifications.

Another aspect of formal languages is the choice of executable versus non-executable specification languages. Both types have their merits. We chose the non-executable language *Z* because it *forces* modelers to think more abstractly by removing the all too familiar programming level terms from their use. When using executable, wide spectrum languages like that provided with the Software Refinery environment [6], one can still rely on these lower-level concepts. However, one should not get the impression that *Z* is completely a "pencil and paper" language. Some tools do exist for syntax checking, type checking, and pretty-printing the *Z* specifications. While more sophisticated tools for *Z* are under development, the only other tool we would like to have is a general-purpose theorem prover.

The approach described in this report has been taught in an introductory course in Software Engineering at the Air Force Institute of Technology. Overall, this course has been very successful at simultaneously introducing students to formal methods and object-oriented modeling. Introducing the mathematics and *Z* formalism at the beginning of the course and integrating the formalism along with the informal model adds to the sense of purpose of the formalism. Combined with our advanced course in formal-based methods, this approach produces a very enlightened set of software engineers.

2 Informal Object Oriented Analysis Model

The basic informal object oriented analysis (OOA) model for this work is based on that of Rumbaugh, et. al.[7]. This was chosen for several reasons. Along with that of Coad and Yourdon [9] it is one of the most comprehensive object oriented analysis models published. It is well recognized in the object oriented literature. In addition, it was already in use as a text for our object oriented modeling course.

Rumbaugh's model consists of three parts: the *object model*, which captures the structural properties of objects and their relationships to each other, as well as a set of operations for each object; the *dynamic model*, which captures the control aspects of the object which change over time; and the *functional model*, which captures the transformation of data values within a system of objects. These three models are represented by variations of the traditional graphical models of entity-relationship (E-R) diagrams, state transition diagrams (STDs), and data flow diagrams (DFDs) respectively. Each of these models is augmented by a data dictionary describing the models in more detail using natural language or any other desired specification language.

The models as introduced by Rumbaugh can be used to support all phases of the software life cycle from requirements analysis through implementation. This section of the paper describes the subset of the Rumbaugh model used as the basis for our formal extensions.

2.1 The OOA Object Model

The object model captures the static structural properties of objects, and their relationships to each other. Important artifacts include objects (along with their attributes) and associations, where special categories of the latter include aggregation and inheritance.

2.1.1 Objects

The object model consists of a set of related *objects*. Each is defined by a set of *attributes*, the values of which define the state of the object. We do not explicitly differentiate between those attributes which remain constant over time (*e.g.* a person's birth date) and those that can change (*e.g.* a person's weight). Objects are defined based on the problem space, not the solution space, so that their attributes should reflect actual attributes of the object. Referential attributes (those that refer to another object, such as a person's spouse's name) should not be included, but in the OOA model should be captured as a relationship between two persons. While such treatment may not be efficient, we prefer to defer efficiency issues to the design or implementation phase. However, at times such an approach seems extreme, as in a person's birth date being a relationship to a "day" object. Thus the analyst has some latitude of decision, based on the level of abstraction and expected purpose for the model. On the other hand, we stress model reuse, even in the analysis phase, which is better supported by keeping object models as "pure" as possible.

The object models at this stage represent *object classes*. Each object definition is a template that represents any actual objects of that class. Thus the definition of a "person" object defines the attributes held by any *instance* of the object such as "Mary Jones" or "Tom Smith." In this paper the terms *class*, *object* and *object class* are used interchangeably, and the term *object instance* (or just *instance*) will be used when dealing with instances. Although for some objects a single instance exists in the system, handling this difference is left as a design or implementation issue. Objects are represented on the object (E-R) diagram by a rectangle containing the object class's name.

We support Rumbaugh's concept of object instances being *distinguishable* without the need for a unique set of distinguishing attributes (frequently referred to as a *key* in database terminology). We refer to an implied (not explicitly represented) *handle* attribute as a means of distinguishing between instances when such a concept helps explain the lack of a "key field." Another implied attribute consistent with the Rumbaugh model is that of the object's *class*. That is, an object's class is always assumed to be known by the object, even though the class is not explicitly represented by an attribute.

2.1.2 Associations

Relationships between objects are modeled as *associations*. An association, similar to an object class, represents a group of possible relationships between object instances. An actual instance of an association, relating two specific object instances, is termed a *link* by Rumbaugh.

Most associations are binary (*i. e.* they relate two objects) and appear on the object diagram as a line between the two object rectangles. Associations are named, with the name written next to the line. Associations can have attributes themselves (as in the case of a "date" attribute for a "marriage" association between two "person" objects). Special notation is provided to indicate such associations on the object diagram. Ternary associations (relating three objects) are also defined and represented by a diamond with lines to the three object rectangles. Higher order associations are possible, but encountered infrequently in real applications.

Multiplicity defines how many instances of one class are associated with a single instance of the other class. The multiplicity of an association can be 1:1, 1:n, or m:n (many to many). In addition, membership of an object instance in an association may be required or optional. Such factors are represented on the object diagram by a system of filled and hollow circles, numbers, and "+" signs at each end of the association line.

2.1.3 Aggregation and Object Management

One special type of association is called *aggregation* by Rumbaugh. This is sometimes referred to as the "composed-of" or "part-of" association, and allows explicit modeling of one object that is composed of other objects as parts. Recursive aggregation is allowed. A diamond is added to the end of the line connected to the parent, or *aggregate* object, while the part, or *component* end of the line is terminated as in regular associations. An aggregation can represent one or many instances of the component class making up the aggregate. In our approach an aggregate is frequently used to model a group or set of objects, such as an organization of students.

When an object class is specified in the analysis phase, it is implied that any number of instances of that class can exist, and instances can be created and deleted dynamically. In design and implementation, some mechanism, explicit or implicit, is needed to keep track of and allow access to individual object instances. This *object management* is normally not addressed in the analysis phase. Thus object instances can belong to an aggregation set or simply exist in the "ether."

2.1.4 Inheritance

Inheritance is an important part of any true object oriented model. It allows subclasses to be defined which have (inherit) all of the properties (attributes, associations, and operations) of their superclass (parent class). This is a special association (often called the "is-a" association) represented on Rumbaugh's object diagrams by adding a triangle to the association line, with the apex pointing to the superclass. A subclass inherits all of its parent class's attributes and all associations. In addition, a subclass can add additional attributes or associations and can restrict the set of allowable values for an attribute, but cannot eliminate an attribute entirely.

2.1.5 Features Not Included

Some of Rumbaugh's model features are not included in the informal part of our model. Default values and data types of attributes are deferred to the formal extensions. Attributes are not explicitly shown on the object diagram. Operations are not considered for our OOA model. Rather behavior is captured by the dynamic and functional models. The Rumbaugh object model is an extremely rich model, providing explicit features to model many special cases. Among such features not addressed in our approach are: role names; ordering of relations (we can use sequences); qualification (these are captured in the formal schemas); inheritance of operations is deferred to the design phase; propagation of operations in aggregation isn't considered; and partitioning issues (modules and sheets) aren't addressed, but can be applied directly

as defined by Rumbaugh. Other aspects not addressed include candidate keys, derived objects and links, homomorphisms, and explicit designation of abstract vs concrete classes.

2.2 The OOA Dynamic Model

The dynamic model captures those aspects of an object and its associations that change over time. The basic model followed is a traditional state transition diagram. Although some authors (*e. g.* Mellor and Schlaer [10]) define a dynamic model for every object, Rumbaugh restricts the dynamic model to those objects "with interesting behavior."

Artifacts of interest include states, events, activities and actions, and event flow diagrams.

2.2.1 States

In general, an object's state space is defined by the allowable values of all of its attributes, and its actual state at any point of time is defined by the current values of those attributes. The dynamic model represents a partitioning of an object's state space. In Rumbaugh's model each of these partitions is what is referred to as a *state* in an object's dynamic model. Thus, for example, the single attribute "age" for a "person" object might define a hundred or more states for a person. The dynamic model might include the states "childhood," "adolescence," "adulthood," and "old age." Rumbaugh points out that the dynamic model can capture temporary changes of an object's type. Modeling "adult" and "old person" as subclasses of "person" would be incorrect since an object can't change its class. Thus "inherent differences among objects are therefore properly modeled as different classes, while temporary differences are properly modeled as different states of the same class." [7]

A state then represents a partition of the object's state space that represents unique behavior with respect to the other states. Many objects turn out to be *passive* objects, in Rumbaugh's terminology – those that simply reply to requests for values of their attributes. For these no state diagram is required. A state is entered based on some (usually external) event (see Section 2.2.2), and upon entering a state some activity is (usually) performed (see Section 2.2.3).

States are named, and represented on a state transition diagram by ovals. States can be hierarchically decomposed into substates. The resulting nested state diagrams help overcome the problem of state explosion often cited as a disadvantage of state modeling. Directed arcs between the state ovals represent allowable *transitions* between states, caused by events. Thus a state can be viewed as an object's response to an event. A state is defined to have duration over time. An object can only be in one state at a time. However, such a state can be a combination of states in "concurrent" dynamic models. For example, a person could be in the "adulthood" state of the dynamic model described above, and also in the "middle class" state of a dynamic model based on a person's "salary" attribute.

2.2.2 Events

Events cause the transition between states for an object, and are written as labels on the corresponding state diagram transition arcs. In Rumbaugh's model such events are defined to originate external to the object, and are in fact generated, or "sent" by another object. (An object *can* cause a transition between its *own* states; this is represented by an unlabeled arc between the two states). An event occurs instantly, at a point in time. An event can carry *attributes* or *parameters*. As with objects and associations, events can represent a class or a specific instance, although no separate terminology is used here.

Since an object can only be in one state at a time, the events causing transition out of a state must be mutually exclusive (otherwise the object would transition to two or more states when such an event occurred). The same event can appear on more than one transition arc, making the state entered dependent on both the event itself and on the state in which the event occurred. Although self-looping transitions are allowed, the assumption is that if no arc labeled for event "E" is shown exiting a state, then the object would not respond to event "E" when in that state.

Transitions (either event-driven or unlabeled) can be constrained by Boolean *guard conditions.* Thus two arcs out of the "running" state of a motor might be labeled with the same event, but with one conditioned by "safety switch ON" and the other by "safety switch OFF." The guard condition must evaluate to TRUE when the event occurs for the transition to take place.

2.2.3 Activities and Actions

Behavior associated with a state is defined as an *activity.* This is similar to the Moore version of state models, where an output is associated with a state. An object begins "execution" of an activity upon entering a state. This can represent a continuous action, such as turning on an alarm, that continues until another event causes the object to leave the state, or a sequence of events that are performed. In the latter case the object may automatically leave the state when the activity is complete, or may wait at the end for an external event. If an event occurs before such a sequence is completed, the activity is interrupted.

An *action* is associated with a transition (and hence with an event). This is similar to the Mealy version of state models, where an output is associated with a transition. Since events and transitions are considered instantaneous, an action is viewed as taking no time, such as triggering an alarm or sending an event to another object.

Although Mealy and Moore machines are alternate modeling approaches, both activities and actions are allowed in the same Rumbaugh dynamic model, providing a rich set of modeling possibilities. Activities can be defined in a variety of ways including natural language text, pseudo code, or a predicate logic statement of the conditions prevailing while in the state, and would typically be detailed, along with other characterizations of the state, in the data dictionary. Actions are named or described on the state transition diagram itself.

2.2.4 Event Flow Diagram

An *event flow diagram* is a useful way of documenting which objects must communicate with each other. Objects are represented by rectangles, with a directed arc connecting an object that sends events to the object which receives those events. The arc is labeled with a list of the events that can be sent.

2.3 The OOA Functional Model

The functional model consists of traditional data flow diagrams (DFDs) used for the classical purpose of diagramming the flow of data between processes. The functional model complements the dynamic model in specifying the behavior of the system.

The DFDs consist of circles ("bubbles") representing *processes* or calculations to be done, connected by directed arcs representing *data flows* between the processes. The data is used in or produced by calculations in the processes. In addition *data stores* are defined for the storage of information, and are represented on the diagram by a pair of horizontal parallel lines. *Actors* represent external sources and sinks of the data flows, and are represented by squares. The processes can be functionally decomposed into lower level DFDs. The actual calculation done by each process is specified in the data dictionary as a process description. This can take many forms, such as pseudo code or pre- and post-conditions. Both Rumbaugh [7] and Yourdon [11] state that such process descriptions should only be developed for the lowest level, or *leaf* processes. We follow that convention. The DFDs are meant to capture the functional specification of the system's behavior in the sense of what calculations are to be performed, along with the sources and destinations of the input and output data of each calculation. The control aspect is captured by the dynamic model in the sense of when calculations are to be performed (i.e. which state's behavior is appropriate).

Although the use of the functional model during the analysis model development is somewhat unclear in Rumbaugh, we find the best approach is to develop a DFD for each active object, that is, each object class for which a dynamic model was developed. Passive objects only respond to queries to read or write attribute values, and have no dynamic model. However, if a passive object has a *derived attribute*, particularly one

with complex calculations or one that depends on values from other objects, then a DFD is also developed for that object.

In the DFD for a given object, any access (get or put) to that object's attribute values is represented by a data store with that object's name. Any passive objects which act as sources or sinks of data flows on the DFD are also represented as data stores. Process bubbles that simply define reading or writing a data store are not explicitly shown.

Exchange of data with other active objects is handled in two ways. If such data is a parameter of an event (either from the other object or to the other object), the object is shown as an actor. The same actor may appear several times to simplify the drawing. However, if the data flow represents a get or put operation of the other object's attribute values, then the other object is represented as a data store. Thus the same external object may appear both as an actor and as a data store on the same DFD.

Finally, if data is to be stored or retrieved from an association (including determining membership), then the association is shown as a data store. Note that such a data store should not appear on a DFD unless the object class is a member of that association.

At the object level, there will typically be one process bubble per state in the dynamic model, or one for each activity within a state, as well as one for each action (either in a state or on a transition) that is more than just sending an event to another object. However, the arcs connecting the bubbles on a DFD represent the flow of data, along with its sources and destinations, while the arcs connecting the states in the dynamic model represent control flow between these processes. A complex state activity (process) can then be further decomposed in the functional model using the DFDs.

2.4 System Level Considerations

We view the system being specified as an object itself, composed of all the other objects. Thus the "system object" is associated with all other objects using aggregation. Since aggregation is itself hierarchical, only the top level aggregate of any complex object needs be connected to the system object with an explicit aggregation line. Since this is an analysis model, it is important to include only problem space objects in the system model.

3 The Syntax of *Z*

This section discusses the essential aspects of the *Z* syntax. The reader is referred elsewhere for more detail [1] [2].

3.1 Z Schemas

In *Z*, most specifications are in the form of *schemas*. A schema is a boxed-in definition consisting of a *schema name*, a *signature*, and a *predicate*. *Static* schemas are used to define the state space of a system, while *dynamic* schemas are used to specify operations on the state space.

3.1.1 Static Schemas

The signature portion of a static schema declares the system's state variables, each defined over a *type* (discussed in Section 3.2 below). The predicate portion defines *invariants* or conditions that hold over the schema's variables at all times. For example, a person might be described as follows.

People

$$name : \text{seq}\, CHAR$$
$$age : \mathcal{N}$$
$$sex : \{M, F\}$$

$$age > 0$$

Schema *inclusion* consists of including the name of one schema in the signature of another, as follows.

Employee

$People$
$employee\_number : \mathrm{seq}\, CHAR$
$salary : \mathcal{R}$

$salary > 0.0$

All of the variable declarations and predicates of the included schema become part of the new schema. (This is explained in more detail in Section 3.4.1). It should be noted that a schema can be defined with no explicit predicate. The implicit predicate in such cases is TRUE (when conjuncted with an included predicate).

3.1.2 Dynamic Schemas

A dynamic schema is used to specify an operation on a system. The signature portion *includes* a copy of the system's state variables before the operation and a copy (decorated with a " $'$ ") after the operation. In addition, new variables can be declared, including *inputs* (decorated with a "?") and *outputs* (decorated with a "!"). The predicate portion defines preconditions (predicates involving "before" variables) and postconditions (predicates involving "after" variables). For example, adding a raise to an employee's salary might be specified as follows.

SalaryRaise

$Employee$
$Employee'$
$raise? : \mathcal{N}_1$
$new\_salary! : \mathcal{N}_1$

$salary' = salary + raise?$
$new\_salary! = salary'$

A shorthand notation uses $\Delta SchemaName$ as follows.

SalaryRaise

$\Delta Employee$
$raise? : \mathcal{N}_1$
$new\_salary! : \mathcal{N}_1$

$salary' = salary + raise?$
$new\_salary! = salary'$

3.2 Declaration of Types

In a schema signature, variables are declared over *sets* or basic types. These types can be defined several ways. There are some standard accepted sets, such as $\mathcal{N}$, $\mathcal{N}_1$, $\mathcal{R}$, $\mathcal{Z}$. Also, sets can be defined by declaring their name in square brackets as follows before using them in schema declarations.

$[TIME]$

$[DATE, MONEY]$

These represent sets whose elements are not modeled in any further detail.

Sets can be declared globally using axiomatic definitions, as with YEAR in Section 3.3 below. Enumerated sets can be handled two ways. They can be declared before use as a *data type definition.*

$DOOR\_STATE ::= open \mid closed$

$LOCK\_STATE ::= locked \mid unlocked$

Access

$door : DOOR\_STATE$
$lock : LOCK\_STATE$

$lock = locked \Rightarrow door = closed$

Alternately, enumerated sets can be shown explicitly in the schema.

Access

$door : \{open, closed\}$
$lock : \{locked, unlocked\}$

$lock = locked \Rightarrow door = closed$

3.3 Globals

Definitions and predicates can be made globally by the use of *axiomatic descriptions.*

$maxcount : \mathcal{N}_1$

$limit : \mathcal{N}_1$

$limit \leq 1000$

$YEAR : \mathrm{seq}\, DIGITS$

$\#YEAR = 4$

3.4 Schema Usage Syntax

3.4.1 Schema Inclusion

As discussed in the previous section, schema inclusion involves using one schema name in the signature of another. The result is to union the signatures and conjunct the predicates. Consider the *People* schema and *Employee* schema defined in Section 3.1.1.

People

$name : \mathrm{seq}\, CHAR$
$age : \mathcal{N}$
$sex : \{M, F\}$

$age > 0$

Employee

People
employee_number : seq $CHAR$
salary : $\mathcal{N}$

---

salary > 0.0

Forming the union of the signatures and the conjunction of the predicates yields the equivalent definition as follows.

Employee

name : seq $CHAR$
age : $\mathcal{N}$
sex : $\{M, F\}$
employee_number : seq $CHAR$
salary : $\mathcal{N}$

---

age > 0
salary > 0.0

3.4.2 Schemas as types

A previously defined schema can be used as a type in a new schema. Although this technique is mentioned by several authors, the syntax is not well defined and few examples can be found. However, this usage of schemas is critical to our application to object oriented systems. Spivey [3] in Section 3.3 states "When a schema name has been defined, ... it can be used in a schema reference to refer to a schema. A schema reference can be used as a declaration, ..." On page 65 he states "A schema reference may be used as an expression: its value is the set of bindings in which the values of the components obey the property of the schema. The schema reference S used as an expression is equivalent to the set comprehension $\{S \bullet \theta S\}$." On page 72 he states "A schema reference S' may be used as a predicate: it is true in exactly those situations which, when restricted to the signature of the schema, satisfy its property. It is effectively equivalent to the predicate $\theta S' \in S$, where S as an expression means $\{S \bullet \theta S\}$." Finally on page 145 he states "The syntax of set expressions is ambiguous: if S is a schema, the expression $\{S\}$ may be either a (singleton) set display or a set comprehension, equivalent to $\{S \bullet \theta S\}$. The expression should be interpreted as a set comprehension; the set display can be written $\{(S)\}$."

Hayes [4], on page 17 in his glossary says "When a schema name S is used as a type it stands for the set of all objects described by the schema, {S}, e.g., w:S declares a variable w with components x (a natural number) and y (a sequence of natural numbers) such that $x \leq \#y$." where S is defined by

S

$x : \mathcal{N}$
y : seq $\mathcal{N}$

$x \leq \#y$

On page 17 he defines *tuple S* as "The tuple formed from a schema's variables: e.g., **tuple S is (x,y).** *Where there is no risk of ambiguity, the word 'tuple' can be omitted, so that just 'S' is written for '(x,y)'*" (emphasis added).

Ince [2] on page 267 states "the schema ... when enclosed in curly brackets {S} is equivalent to the comprehensive set specification $\{x : \mathcal{N};\ y :\ \text{seq}\ \mathcal{N} \mid x \leq \#y\}$ which is the set of ..." (example schema S substituted for his example), but does not show such a use inside another schema's signature.

Finally, Sommerville [5] in Figure 9.12 shows the following schema.

DataDictionary

$$\begin{array}{|l} DataDictionaryEntry \\ ddict : NAME \pfun \{DataDictionaryEntry\} \end{array}$$

where NAME is a type definition, and DataDictionaryEntry is a schema. Function ddict maps a NAME to a DataDictionaryEntry. Based on the definitions of Spivey and Hayes, it would seem the proper syntax should be as follows.

DataDictionary

$$\begin{array}{|l} DataDictionaryEntry \\ ddict : NAME \pfun DataDictionaryEntry \end{array}$$

He also shows the following schema in Figure 9.21.

NewDataDictionary

$$\begin{array}{|l} DataDictionaryEntry \\ ddict : \mathrm{seq}\{DataDictionaryEntry\} \\ \hline \forall i, j : \mathrm{dom}\, ddict \bullet (i < j) \Rightarrow s(i).ident <_{...NAME} s(j).ident \end{array}$$

with the implication that DataDictionaryEntry must be *included* in order to refer to s.ident, a component of DataDictionaryEntry. This is not illustrated in any of the other authors' examples.

We interpret the syntax as follows. Consider the case for a (traditional) family consisting of two persons, a husband and a wife, with several children, where only children under the age of eighteen are included. In addition, a set of chores to be performed by family members are defined.

$[JOBS]$

Family

$$\begin{array}{|l} husband : People \\ wife : People \\ children : \mathbb{P} People \\ chores : JOBS \rel People \\ \hline husband.age \geq 18 \\ wife.age \geq 18 \\ husband.sex = M \\ wife.sex = F \\ \forall x : children \bullet x.age < 18 \\ \mathrm{dom}\, chores = JOBS \\ \mathrm{ran}\, chores = children \cup \{husband\} \cup \{wife\} \end{array}$$

The syntax here represents that *husband* and *wife* is each a variable of type People, while *children* is a set of People. The relation *chores* is a set of ordered pairs, each of which consists of a JOB and a People. Since it is a relation, each job can be assigned to one or more persons and each person can be assigned more than one job.

The first four predicates simply put constraints on attribute values of family members, and illustrate the use of the "." syntax for referring to components of a schema type (e.g. wife.age). Our convention is

that a schema doesn't have to be *included* in order to access its variable types if it has been used in a type declaration.

The fifth predicate specifies that all members of the *children* set must have an age of less than eighteen. The sixth predicate indicates that every job must be assigned a person in the *chores* relation, while the last predicate specifies that every member of the family must be assigned at least one job.

3.4.3 Schema Calculus

Schemas can be combined and manipulated in more sophisticated ways. The reader is referred to other sources for more detail [3] [1] [2] [4].

4 *Z* Extensions to the OOA Model

The formal extensions discussed in this section are built on the object oriented analysis model of Rumbaugh, et. al.[7]. All three parts of Rumbaugh's model, the *object model*, the *dynamic model*, and the *functional model*, are included. Static *Z schemas* are used to specify objects and associations in the object model as well as to define states in the dynamic model. Dynamic *Z schemas* are used to specify processes in the functional model. In addition, a state transition table, using formal syntax, is used to define transitions in the dynamic model.

4.1 The Extended OOA Object Model

The object model captures the static structural properties of objects and their relationships to each other. Both objects and associations are formalized, including the special association categories of inheritance and aggregation.

4.1.1 Extended Objects

The object model is formalized by defining a *Z* static schema for each object class. The schema is named the same as the object, where by convention we capitalize the first letter of the name. The signature portion of the schema defines each attribute by its name (by convention spelled with a lower case letter) and a set-theoretic type over which it is defined.

We continue Rumbaugh's concept of object instances being *distinguishable* without requiring a unique attribute by treating all attributes the same in the schema signature. The implicit *handle* attribute as well as the object's *class* are not included in the schema. (Other researchers have taken the approach of requiring a unique instance identifier attribute and using this as the basis of defining a *function* relating it to the other attributes [12]).

Another concept of Rumbaugh's that we continue is that attributes represent pure, atomic values, and not other objects. Thus the set-theoretic data types used in object schemas are predefined types. These are normally expressed in all upper case, and represent predefined sets or sets defined globally as axioms. An exception to this occurs when dealing with aggregation, discussed in Section 4.1.4.

The *predicate* portion of the schema contains Boolean predicates that represent *invariants* for the object; that is, assertions that must be true at all times. This is similar to Rumbaugh's notion of constraints, but here consists of invariants on and between the object's attributes. Predicates can also used to specify *derived attributes*, that is, attributes that can be derived from other attributes of the object. Although these could be omitted, the derived attributes turn out to be useful when "connecting" inputs to outputs in an aggregate model (see Section 4.1.4).

Consider as an example the following informal model of a *vehicle* class.

Vehicle
vehicle type: vehicle_set
serial number: string
model type: string
model year: integer
weight in kg: integer
primary color: color_set
second color color_set
max speed in kph: integer

The corresponding Z schema would be:

$[VEHICLES, CHAR, MODELS, COLORS, DIGITS]$

$$YEAR : \text{seq}\, DIGITS$$
$$\#YEAR = 4$$

Vehicle

$$veh\_type : VEHICLES$$
$$serial\_number : \text{seq}\, CHAR$$
$$model\_type : MODELS$$
$$model\_year : YEAR$$
$$weight\_kg : \mathcal{N}$$
$$primary\_color : COLORS$$
$$second\_color : COLORS$$
$$max\_speed\_kph : \mathcal{N}$$
$$weight\_kg > 0$$
$$max\_speed\_kph > 0$$

As another example, consider the Z specification for a *person* object class, to be used in later examples. Note the use of the globally defined $TODAY$ to represent the current date and the derived attribute *age*.

$[CHAR, DATE, DIGITS]$

$SEXTYPE ::= male \mid female$

$$SSAN : \text{seq}\, DIGITS$$
$$\#SSAN = 9$$

$$TODAY : DATE$$
$$\exists d : DATE \bullet d = TODAY \wedge TODAY \text{ is the current day.}$$

$$yearinterval : DATE \times DATE \rightarrow \mathcal{Z}$$
$$\forall d_1, d_2 \in DATE \bullet yearinterval(d_1, d_2) \text{returns the number of complete years from } d_1 \text{ to } d_2.$$

$$
\begin{array}{|l}
\hline
Person \\
lastname : \mathrm{seq}\, CHAR \\
initial : CHAR \\
firstname : \mathrm{seq}\, CHAR \\
birthdate : DATE \\
ssan : SSAN \\
sex : SEXTYPE \\
age : \mathcal{N}_1 \\
\hline
age = yearinterval(birthdate, TODAY) \\
\hline
\end{array}
$$

4.1.2 Extended Associations

Relationships between objects are modeled as *associations.* Objects are represented on the object diagram by a rectangle containing the object class's name. Binary associations appear on the object diagram as a line between the two related object rectangles, with the association name written next to the line. In general, each binary association is documented as a *Z* static schema. Multiplicity of an association can be one to one (1:1), one to many (1:n), or many to many (m:n), and membership of an object instance in an association may be required or optional. Such factors are represented on the object diagram using Rumbaugh's syntax of filled and hollow circles, numbers, and "+" signs at each end of the association line. In our extended model, these define the form of the association definition.

Each association is represented by a *Z* static schema. The schema name corresponds to the association name with the first letter in upper case. The set theoretic types used in the association schema are the static schemas defining the object types that are related. The signature includes a form of mathematical relation (relation, function, injection, etc. as outlined below) using the name of the schema spelled with a leading *lower* case letter.

For a general m:n association, the signature declaration takes the form of a *relation.* Consider an association *owns* relating a person instance to a vehicle instance, as shown in Figure 1.

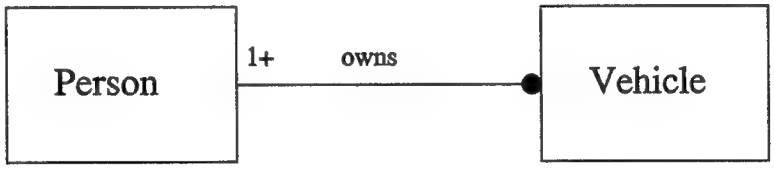

Figure 1: The Association *owns.*

Note by the symbology at the ends of the line that in this model a vehicle can be owned by more than one person, and a person may own more than one vehicle (m:n). Furthermore note that each vehicle must have at least one owner, while a person need not own any vehicles. The corresponding *Z* schema would look as follows.

$$
\begin{array}{|l}
\hline
Owns \\
owns : Person \leftrightarrow Vehicle \\
\hline
\mathrm{ran}\, owns = Vehicle \\
\hline
\end{array}
$$

In this model *owns* is defined as a set of ordered pairs. The set-theoretic data types used in defining the relation are themselves schemas, as outlined in Section 3.4. The predicate allows us to specify that all possible vehicles must have an owner. The predicate "dom $owns = Person$" would be added if every person was required to own a vehicle, and neither predicate would be included if membership of both sets in the

association were optional. The requirement for these predicates is unique to m:n relations, as the optionality is captured by the specified form of the function in the remaining cases.

Consider the case where each vehicle must have a single registered owner. Then the association becomes 1:n and can be represented as follows.

OneOwns

$$one\_owns : Vehicle \rightarrow Person$$

Here the symbol $\rightarrow$ indicates a total function, which captures the invariant that every vehicle must have an owner. The *Z* syntax includes functions to cover all other cases of multiplicity and membership. These are summarized in Table 1.

Table 1: Multiplicity Representation

Multiplicity	Domain Membership	Range Membership	Formal Specification	zed.sty Symbol	Alternate Symbol
m:n	either	either	Relation w/ predicates	$\leftrightarrow$	
n:1	optional	optional	Partial Function	$\nrightarrow$	
n:1	required	optional	Total Function	$\rightarrow$	
n:1	optional	required	Partial Surjection	$\twoheadrightarrow$	
n:1	required	required	Total Surjection	$\twoheadrightarrow$	
1:1	optional	optional	Partial Injection	$\mathbb{C}$	$\rightarrowtail$
1:1	required	optional	Total Injection	$\subset$	$\rightarrowtail$
1:1	required	required	Bijection	$\subseteq$	$\rightarrowtail$

Another option is where the association line is terminated with a number. For example, consider the object diagram in Figure 2, where each student must take 2 or more sequences. Assuming that schemas have been defined for *Student* and *Sequence* we can specify the association *Takes* as follows.

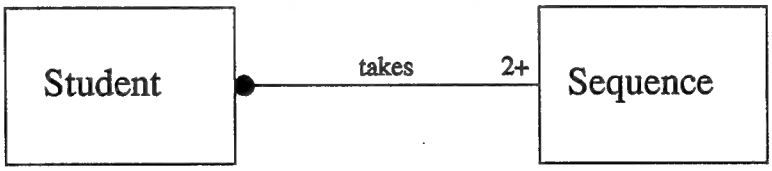

Figure 2: The Association *takes*.

Takes

$$takes : Student \leftrightarrow Sequence$$

$$\forall s : \mathrm{dom}\, takes \bullet \#(\{s\} \lhd takes) \geq 2$$

Associations can have attributes themselves (as in the case of a "date" attribute for a "marriage" association between two "person" objects). We treat all such cases as *associative objects* and define a separate schema for the associative object.

For example, suppose in the earlier example of each vehicle being owned by a single person that such an association is characterized by a "registration number" and "registration date." These are attributes of the association, not of the person or the vehicle alone. Using *Z* this would be expressed as follows.

Registered_Attr

$reg\_number : \text{seq}\, CHAR$
$reg\_date : DATE$

and

Registered

$registered : (Vehicle \rightarrow Person) \rightarrow Registered\_Attr$

Ternary associations (relating three objects) are handled in a similar fashion. Consider a ternary association "drives" that represents a person driving a specific car in a particular race. Assuming that a schema *Race* has been defined, we could specify the ternary association as follows.

Drives

$drives : \mathcal{P}(Person \times Vehicle \times Race)$

4.1.3 Inheritance

Inheritance is a special type of association and is treated differently. Since an instance of the subclass "is-a" instance of the superclass, we first define the superclass schema, then *include* it in the subclass schema. For example, consider a military officer as a specialization of a person. Using the *Person* schema already defined, we could specify the following.

$[RANKSET, BRANCHES, SPEC\_CODES]$

Officer

$Person$
$rank : RANKSET$
$branch : BRANCHES$
$specialty : SPEC\_CODES$

Similarly

$[MAJORS, DATE]$

Student

$Person$
$major : MAJORS$
$minor : MAJORS$
$entered : DATE$
$gpa : \mathcal{R}$

$gpa \geq 0$
$gpa \leq 4.0$

Multiple inheritance can then be specified, as in the case of a military student.

$[CODES]$

MilitaryStudent

$Student$
$Officer$
$ed\_code : CODES$

4.1.4 Aggregation

Since aggregation is just a special type of association, it could be represented as any general association. However, we find it convenient to adopt a slightly different approach.

Collections The simplest aggregate object is one composed of a set of objects of the same type. This occurs frequently in specifying systems. For example, consider a roster of drivers, which could be specified as follows.

Roster

$roster : \mathcal{P} Person$

Similarly, a set of vehicles could be specified.

Fleet

$fleet\_name : \mathrm{seq}\, CHAR$

$fleet : \mathcal{P} Vehicle$

In this last example, a fleet consists of a fleet name and a set of vehicles.

Composition Now consider a transportation company that consists of a roster of drivers and several fleets of vehicles.

TransportCo

$drivers : Roster$

$inventory : \mathcal{P} Fleet$

Or, in a somewhat simpler approach,

TransportCo

$drivers : \mathcal{P} Person$

$inventory : \mathcal{P} Fleet$

Note that in all cases the aggregate association is represented as variable or set declared over the appropriate schema.

As another example, suppose the following three schemas are defined.

Engine

$cylinders : \mathcal{N}_1$

$displacement : \mathcal{N}_1$

$horsepower : \mathcal{N}_1$

$cylinders \geq 4$

Wheel

$diameter : \mathcal{N}_1$

$width : \mathcal{N}_1$

$pressure : \mathcal{N}_1$

$$
\begin{array}{|l}
\hline
WheelSet \\
\hline
wheels : \mathcal{P} Wheel \\
\hline
\end{array}
$$

Then the vehicle specification could be defined as follows.

$$
\begin{array}{|l}
\hline
Vehicle \\
\hline
engine : Engine \\
all\_wheels : WheelSet \\
veh\_type : VEHICLES \\
serial\_number : \mathrm{seq}\, CHAR \\
model\_type : MODELS \\
model\_year : YEAR \\
weight\_kg : \mathcal{N} \\
primary\_color : COLORS \\
second\_color : COLORS \\
max\_speed\_kph : \mathcal{N} \\
\hline
weight\_kg > 0 \\
max\_speed\_kph > 0 \\
veh\_type =' auto' \Rightarrow \#all\_wheels.wheels = 4 \\
veh\_type =' cycle' \Rightarrow \#all\_wheels.wheels = 2 \\
\hline
\end{array}
$$

Associations Between Components Since an aggregate object represents an object in and of itself, we find it useful to embed associations between its component objects in the schema definition of the aggregate. Consider the transportation company defined earlier, which consisted of a roster of drivers and a set of vehicle fleets.

$$
\begin{array}{|l}
\hline
TransportCo \\
\hline
drivers : Roster \\
inventory : \mathcal{P} Fleet \\
\hline
\end{array}
$$

Suppose that each driver is assigned to a fleet. Fleet assignments are not permanent and can be changed by the dispatcher, but a driver can only be assigned to one fleet at a time. Obviously a fleet can have many drivers assigned, but under some circumstances a fleet might have no drivers assigned, and at times a driver might not be assigned to any fleet. Since schedules are made out in advance, and a driver could be assigned to the same fleet for different intervals, each assignment has a specified effective date. Since the Assignment association only makes sense in the context of the transportation company, we prefer to define it as part of the aggregate object schema definition. This allows the relationship (a function in this case) to be expressed directly in terms of component sets as follows.

$$
\begin{array}{|l}
\hline
TransportCo \\
\hline
drivers : Roster \\
inventory : \mathcal{P} Fleet \\
assigned : (drivers.roster \nrightarrow inventory.fleet) \rightarrow DATE \\
\hline
\end{array}
$$

This is somewhat inconsistent with our approach of defining associations in separate static schemas. The only disadvantage to this approach is that it limits the reuse of association specifications.

Pure Aggregates Rumbaugh is vague on whether an aggregate can exist as an instance with no components. There may be an advantage in formalizing OOA to define (or require) the idea of a *pure aggregate.*

Such an aggregate would have no attributes of its own. (A variant would allow static attributes, such as a vehicle serial number, which realistically is not associated with any one component). This is in line with the SEI philosophy [13] [14] that all object hierarchies should be "flat," that is have only a single level of hierarchies, and that all lower level aggregates are model artifacts that are not themselves real.

The above example might then become as follows.

$$
\begin{array}{|l}
\hline
Vehicle \\
engine : Engine \\
all\_wheels : WheelSet \\
body : Body \\
\hline
\end{array}
$$

Note that primary_color and second_color are attributes of "Body." Also weight_kg and max_speed_kph can be derived from the aggregation of attributes of the components. Attributes veh_type, model_type, and model_year are class attributes of a subclass and could possibly be handled through specialization or by a pseudo-model. Only serial_number presents a problem of being an attribute of an instance of the aggregate instance itself.

4.2 The Extended Dynamic Model

The dynamic model uses a traditional state transition diagram to capture those aspects of an object that change over time. Our extended model includes states, events, activities and actions, and event flow diagrams.

4.2.1 States

An object's state space is defined by (1) the allowable values of all of its attributes, (2) by its membership (or lack of) in those associations in which it can participate, and (3) by the link attribute values of such associations. A state in the dynamic model represents a partition of the object's state space that represents unique behavior with respect to the other states. Thus a "Person" could be in a "child" or "adult" state based on its "age" attribute; it could be in the "employed" or "unemployed" state based on its membership in the "WorksFor" association; and if employed could be in the "MiddleClass" or "Wealthy" state based on the value of the link attribute "salary" of the "WorksFor" association. Note that a person can move between "child" or "adult" independently of moving between "unemployed" or "unemployed," demonstrating the idea of concurrent state partitions for an object.

In our extended model each state is given a name, and a corresponding Z schema is defined. However, a basic static schema is inappropriate, because predicates there define invariants for all members of the class that hold for all times, while different instances of an object can be in different states. Thus for a state schema we *declare* an object of the specified type in the signature portion of the schema and *include* any involved associations. In the predicate portion, the partition of state variables that define this state is expressed as a predicate over the state variables of the declared variable using tuple notation (object.attribute) and/or membership of the input variable in the appropriate associations. These reflect invariants of that state.

Consider the *Employee* class defined in Section 3.1.1. There salary was modeled as an attribute of an employee. A better approach might be to define the salary as a link attribute of association *WorksFor*, where we define the *WorksFor* association between the *Person* class and the *Company* class (not previously defined) as follows.

$$
\begin{array}{|l}
\hline
WorksFor\_Attr \\
position\_number : \mathrm{seq}\, CHAR \\
salary : \mathcal{R} \\
\hline
\end{array}
$$

WorksFor

$$employment : (Person \nrightarrow Company) \rightarrow WorksFor\_Attr$$

$$\forall x \in \text{ran}\, employment \bullet x.salary > 0.0$$

Then a series of states can be defined for a *Person* as follows.

Child

$$p : Person$$

$$p.age < 18$$

Adult

$$p : Person$$

$$p.age \geq 18$$
$$p.age < 65$$

Senior

$$p : Person$$

$$p.age \geq 65$$

Unemployed

$$p : Person$$
$$WorksFor$$

$$p \notin \text{dom}\,\text{dom}\, employment$$

Employed

$$p : Person$$
$$WorksFor$$

$$p \in \text{dom}\,\text{dom}\, employment$$

MiddleClass

$$Employed$$

$$\forall c : Company \bullet (employment(p, c)).salary < 100,000.00$$

Wealthy

$$Employed$$

$$\forall c : Company \bullet (employment(p, c)).salary \geq 100,000.00$$

Note that "MiddleClass" and "Wealthy" are in fact substates of state "Employed."

Upon entering a state some activity is (usually) performed. Either this activity must not modify the object's state variables (*i.e.* perform a true function with no side effects) or any such modifications must be within the range of the current state partition in order for the object to remain in the same state. However, it is possible for the object to modify its state variables or association membership such that it actually changes state. This is discussed as an *automatic transition* in the next section.

The behavior of an object while in a state is described in a narrative manner in the dynamic model. It is defined in more detail, using dynamic schemas, in the functional model.

4.2.2 Events and Transitions

Following the approach of Rumbaugh, a *transition* represents the change from one specific state to another. *Events* external to the object cause the transition between states. An event is considered to occur instantly relative to the duration of a state. An event can carry *parameters* which convey data to the object. An event usually causes a transition, but the transition can be defined as conditional on some guard condition. Finally, in Rumbaugh's model, a transition can cause the execution of an action, often the sending of an event to some other object. Thus a transition can be completely characterized by an initial state, an external event (with optional parameters), an optional guard condition, a target state, and an optional action. We find the most effective way of capturing this aspect of the specification formally is through the use of an *event transition table.*

For example, consider the state transition diagram for a Person based on age and salary, shown in Figure 3. The corresponding state transition table is shown in Table 2.

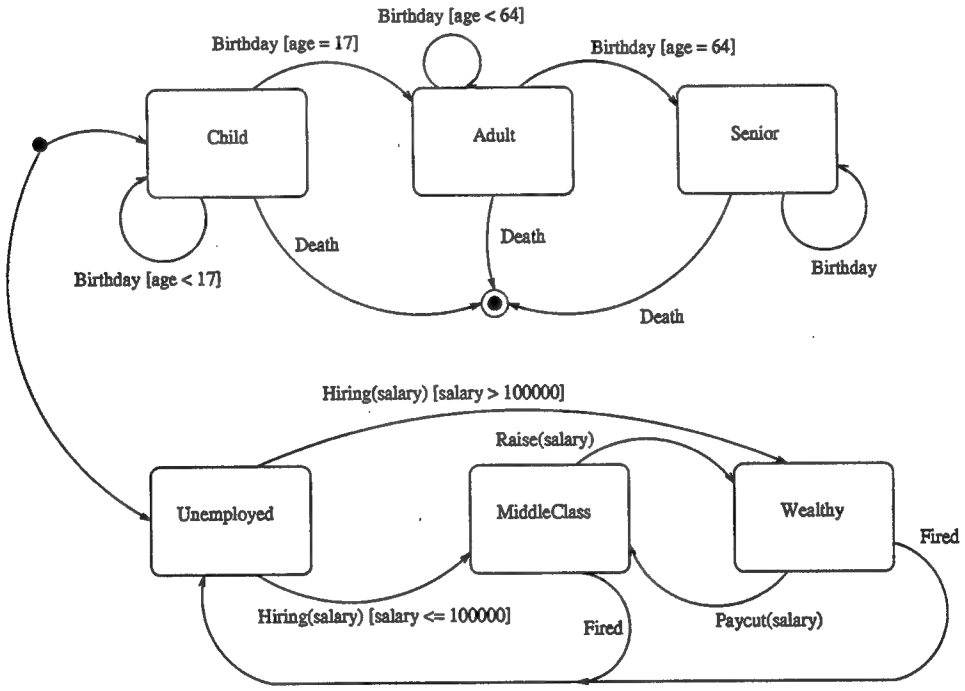

Figure 3: State Transition Diagram for Person.

An *automatic transition* occurs when an object has finished its specified task in a state and transitions to another state without waiting for an external event. This is shown on the state transition diagram as an unlabeled transition arrow (although it *may* have a guard condition and/or an action), and appears in the state transition table with no entry in the "Event" column. Note that this should be used to model an internal transition between two states in which the object's behavior differs, and not used to break up a state's activity into sequential steps.

4.2.3 Event Flow Diagram

An *event flow diagram* is a useful tool for documenting which objects must communicate with each other. In Rumbaugh's syntax objects are represented by rectangles, with a directed arc connecting an object that sends events to the object which receives those events. The arc is labeled with a list of the events that can be sent.

Table 2: State Transition Table.

Current	Event	Parameters	Guard	Next	Action
Child	Birthday		$age < 17$	Child	
Child	Birthday		$age = 17$	Adult	
Child	Death			Terminal	
Adult	Birthday		$age < 64$	Adult	
Adult	Birthday		$age = 64$	Senior	
Adult	Death			Terminal	
Senior	Birthday			Senior	
Senior	Death			Terminal	
Unemployed	Hiring	salary	salary $< 100{,}000$	MiddleClass	
Unemployed	Hiring	salary	salary $\geq 100{,}000$	Wealthy	
MiddleClass	Raise	salary	salary $\geq 100{,}000$	Wealthy	
MiddleClass	Fired			Unemployed	
Wealthy	Paycut	salary	salary $< 100{,}000$	MiddleClass	
Wealthy	Fired			Unemployed	

We find it convenient to use a partial event flow diagram when defining the dynamic model of a single object, showing a single box with an input arrow labeled with a list of external events to which the object can respond, and an output arrow labeled with a list of events which the object can send to other objects, as shown for a Person in Figure 4.

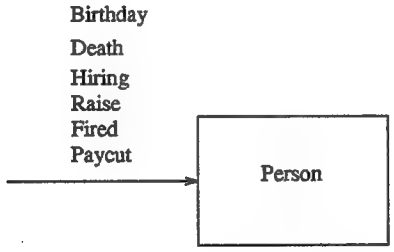

Figure 4: Event Flow Diagram for Person.

4.3 The Extended Functional Model

The extended functional model uses data flow diagrams (DFDs) in the manner specified in Section 2.3. The corresponding process descriptions for the lowest level processes are specified using Z dynamic schemas (Section 3.1.2).

4.3.1 Use of Z

For each leaf bubble, a Z static schema is defined. It *includes* either the corresponding object's Δ schema, in the case that the object's attribute values are modified by the operation, or by the object's Ξ schema, in the case that the object's attribute values are *not* modified by the operation. Inputs that come from other active objects are defined by Z decorated input variables, while outputs that go to other active objects are defined by Z decorated output variables. Access to attributes from other passive objects should be handled by including either the Δ or Ξ schema of the corresponding object or association as needed.

4.3.2 Functional Model Example

Consider the object diagram in Figure 5. This models a course *section* taught by one or more *faculty*

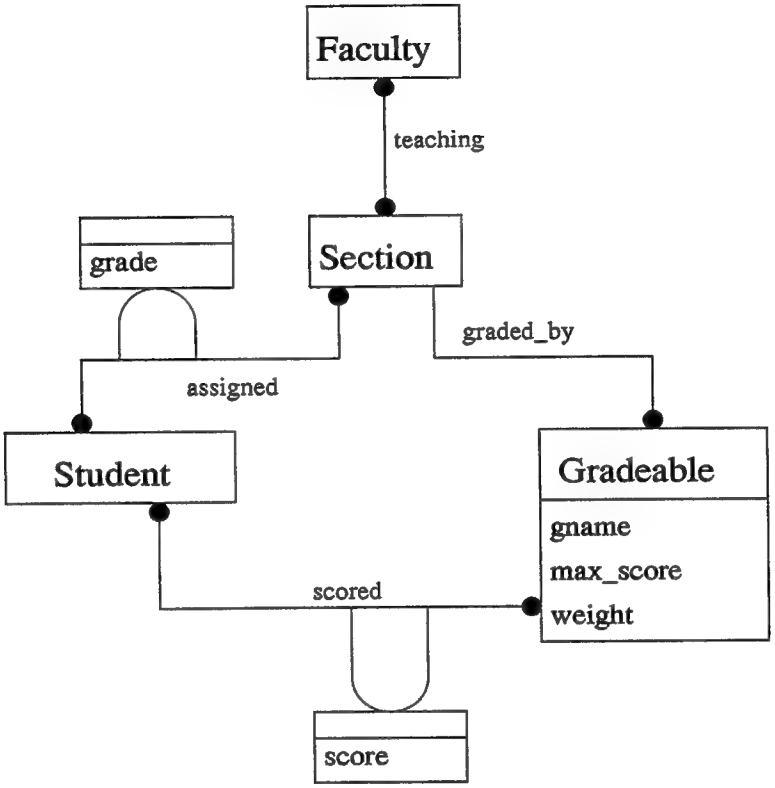

Figure 5: Object Model for Grading System.

and attended by a number of *students*. For each section a number of *gradeables* are defined (homework, exams, projects), with attributes of the gradeable's name, the maximum score that can be attained on that gradeable, and the gradeable's weight in the course. (For a given section, the sum of the weights should be 1.00). Each student is associated with each gradeable via the *scored* association, with its link attribute *score*. The student also has a grade in the course, indicated by the link attribute *grade* on the *assigned* association. The corresponding Z schemas relevant to the functional model are as follows.

Gradeable

$$gname : \mathrm{seq}\, CHAR$$
$$max\_score : \mathcal{R}$$
$$weight : \mathcal{R}$$

$$max\_score \geq 0.0$$
$$weight \geq 0.0$$
$$weight \leq 1.0$$

Graded_by

$$graded\_by : Gradeable \rightarrow Section$$

$$\sum_{\forall g:Gradeable \mid (g,s) \in graded\_by} g.weight = 1.0$$

ScoredAttr

$score : \mathcal{R}$

$score \geq 0.0$

Scored

$scored : (Student \leftrightarrow Gradeable) \rightarrow ScoredAttr$

AssignedAttr

$grade : \{A, A-, B+, B, B-, C+, C, C-, D, F\}$

Assigned

$assigned : (Student \leftrightarrow Section) \rightarrow AssignedAttr$

One of the operations required for this system is as follows. Given a specific section and a gradeable name, calculate the minimum and maximum scores attained on that gradeable, as well as the mean and variance. Figure 6 and Figure 7 show the top level DFD and the top level decomposition respectively.

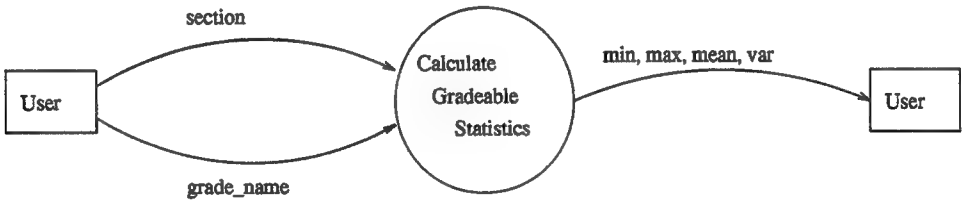

Figure 6: Top Level DFD (Context Diagram).

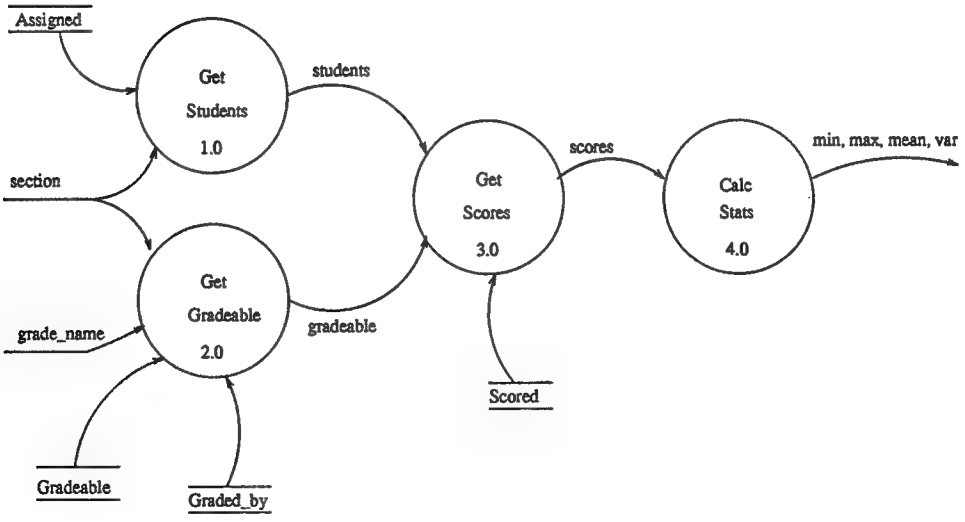

Figure 7: Top Level DFD Decomposition.

In Figure 7, *Get Students*, *Get Gradeable*, and *Get Scores* are leaf processes. They are represented by the following Z schemas.

GetStudents

$\Xi Assigned$
$section? : Section$
$students! : P\, Student$

$students! = \mathrm{dom}((\mathrm{dom}\, assigned) \rhd \{section?\})$

GetGradeable

$\Xi Gradeable$
$\Xi Graded\_by$
$grade\_name? : \mathrm{seq}\, CHAR$
$section? : Section$
$gradeable! : Gradeable$

$\exists g : Gradeable \bullet g.gname = grade\_name? \land (g, section?) \in graded\_by \land gradeable! = g$

GetScores

$\Xi Scored$
$students? : P\, Student$
$gradeable? : Gradeable$
$scores! : (Student \leftrightarrow Gradeable) \rightarrow \mathcal{R}$

$scores! = \{s : Student;\ g : Gradeable;\ n : \mathcal{R} \mid (s, g, n) \in scored \land s \in students? \land g = gradeable?\}$

Figure 8 shows a further level of decomposition of the process *Calc Stats*. This decomposition could probably have been included at the first level, but is shown to illustrate multiple level decomposition.

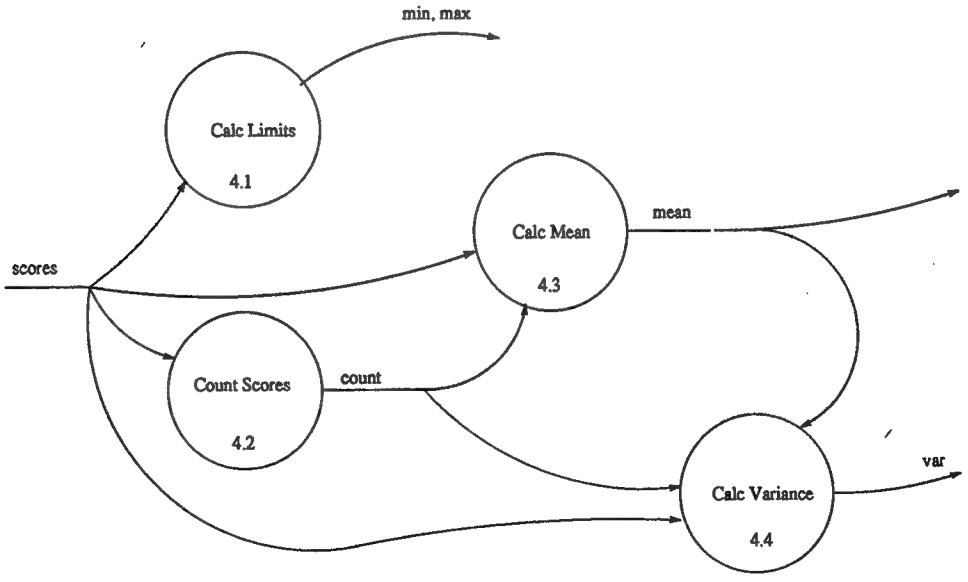

Figure 8: Decomposition of *Calc Stats*.

In Figure 8, *Calc Limits*, *Count Scores*, *Calc Mean*, and *Calc Variance* are leaf processes. They are represented by the following Z schemas.

CalcLimits

$scores? : (Student \leftrightarrow Gradeable) \rightarrow \mathcal{R}$
$min!, max! : \mathcal{R}$

$min! = \min(\text{ran } scores?)$
$max! = \max(\text{ran } scores?)$

CountScores

$scores? : (Student \leftrightarrow Gradeable) \rightarrow \mathcal{R}$
$count! : \mathcal{N}$

$count! = \#scores?$

CalcMean

$scores? : (Student \leftrightarrow Gradeable) \rightarrow \mathcal{R}$
$count? : \mathcal{N}$
$mean! : \mathcal{R}$

$count? > 0$
$mean! = (\sum_{\forall s:\text{dom } scores?} scores?s)/count?$

CalcVariance

$scores? : (Student \leftrightarrow Gradeable) \rightarrow \mathcal{R}$
$count? : \mathcal{N}$
$mean? : \mathcal{R}$
$var! : \mathcal{R}$

$count? > 0$
$var! = (\sum_{\forall s:\text{dom } scores?} (scores?s - mean?)^2)/count?$

4.4 Analysis Methodology

We recommend a *composition* methodology, along the lines of *Z* specification methodologies. That is, individual low-level objects are defined first, using inheritance where appropriate. Then they are combined using aggregation and associations to form more complex objects and systems.

5 Rocket Example

This section presents a simple but complete example of the methodology described in this report. The example is an event-driven simulation of a ground to ground (unguided) rocket. The Simulation System is not modeled here, but is assumed to include a Simulation Clock object and a Next Event Queue object. Each object is assumed to schedule and cancel simulation events with the Next Event Queue. The Simulation System selects events in proper time-stamp order and sends them to the appropriate object at the proper time. Rocket objects have access to the Simulation Clock to read the current simulation time.

The *Rocket* class has subcomponents of a *Fuel Tank* and a *Jet Engine*. There are two of each, with each fuel tank supplying one of the engines. In addition, there is an *Airframe* object that models the flying part of the rocket. Figure 9 shows the overall object model.

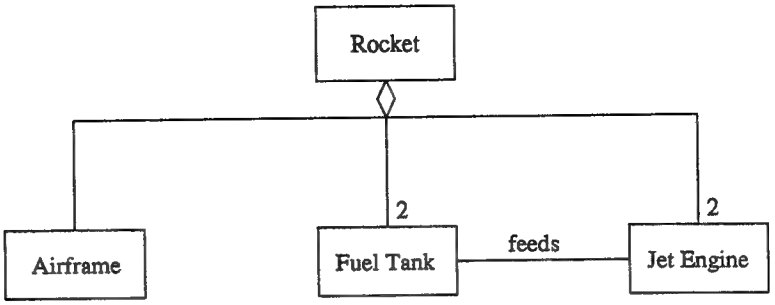

Figure 9: Rocket Object Model.

A *Z*-like bottom up approach is presented here. First the fuel tank, jet engine, and airframe components are defined, then they are combined to form the rocket model.

5.1 Simulation Clock

The model presented in this section is a simple simulation clock needed in the functional models of the other objects. The simulation clock is a passive model. Its sole attribute *sim_time* can be set to zero, read, and written.

5.1.1 The Object Model

Let **SIMTIME** be the set of possible simulation times.

$[SIMTIME]$

SimClock

$sim\_time : SIMTIME$

5.1.2 The Dynamic Model

The simulation clock is a passive object.

5.1.3 The Functional Model

There is no functional model for the simulation clock.

5.2 Next Event Queue (NEQ)

The NEQ is part of the Simulation System. Although its details are beyond the scope of this example, it responds to dynamic events *Schedule(sim_event)* and *Cancel(sim_event)* from Rocket objects, and sends the scheduled *sim_events* to the appropriate object when the simulation clock reaches the sim_event's scheduled time.

5.3 Fuel Tank Model

The model presented in this section is somewhat more detailed than required. It includes the ability to support inflight fueling, and can generate an overflow event. It is used here to demonstrate the concept of reuse at the analysis level.

This model of a fuel tank models the fuel flow aspects of the tank along with the weight of the tank and the fuel. It does not model physical dimensions or shape. Static parameters are the capacity (maximum fuel level) and empty weight of the tank (tank weight), along with the density (weight per unit) of the fuel. State variables include the input flow rate (used in filling) and output flow rate (used in drawing fuel out), and the current fuel level, along with the total weight of the tank and fuel (fuel tank weight). The latter is a *derived attribute* calculated from the empty tank weight, the fuel density, and the fuel level. Units aren't important, as long as they are consistent for all attributes.

The dynamic behavior modeled includes filling the tank, using fuel from the tank, and fueling the tank while fuel is being used. The input and output flow rates are specified by external objects, and the tank sends events to those external objects when it overflows or runs empty. To simplify the model a little, it is assumed that in the *FillAndUse* state the two flow rates are close enough and the time in this state short enough that the tank will neither overflow nor run empty while in this state.

Since this is an event-driven simulation model it is important to keep track of the simulation time at which the state variable values are valid. Thus the *tank_sim_time* attribute maintains the tank simulation time.

5.3.1 The Object Model

The object model consists of the single object *Fuel Tank*, since it has neither components nor a superclass. The *Z* Static Schema is as follows.

$$
\begin{array}{|l}
\textit{FuelTank} \\
\hline
tank\_sim\_time : SIMTIME \\
input\_flow\_rate : \mathcal{R} \\
output\_flow\_rate : \mathcal{R} \\
fuel\_level : \mathcal{R} \\
capacity : \mathcal{R} \\
tank\_weight : \mathcal{R} \\
fuel\_density : \mathcal{R} \\
fuel\_tank\_weight : \mathcal{R} \\
\hline
fuel\_level \leq capacity \\
fuel\_tank\_weight = tank\_weight + (fuel\_density)(fuel\_level) \\
\hline
\end{array}
$$

5.3.2 The Dynamic Model

The dynamic model state transition diagram is shown in Figure 10, and the corresponding state transition table is in Table 3. The partial event-flow diagram is shown in Figure 11.

Empty State In this state the tank is empty, and no fuel is flowing in or out. There are no actions or activities in this state.

$$
\begin{array}{|l}
\textit{Empty} \\
\hline
t : FuelTank \\
\hline
t.fuel\_level = 0 \\
t.input\_flow\_rate = 0 \\
t.output\_flow\_rate = 0 \\
\hline
\end{array}
$$

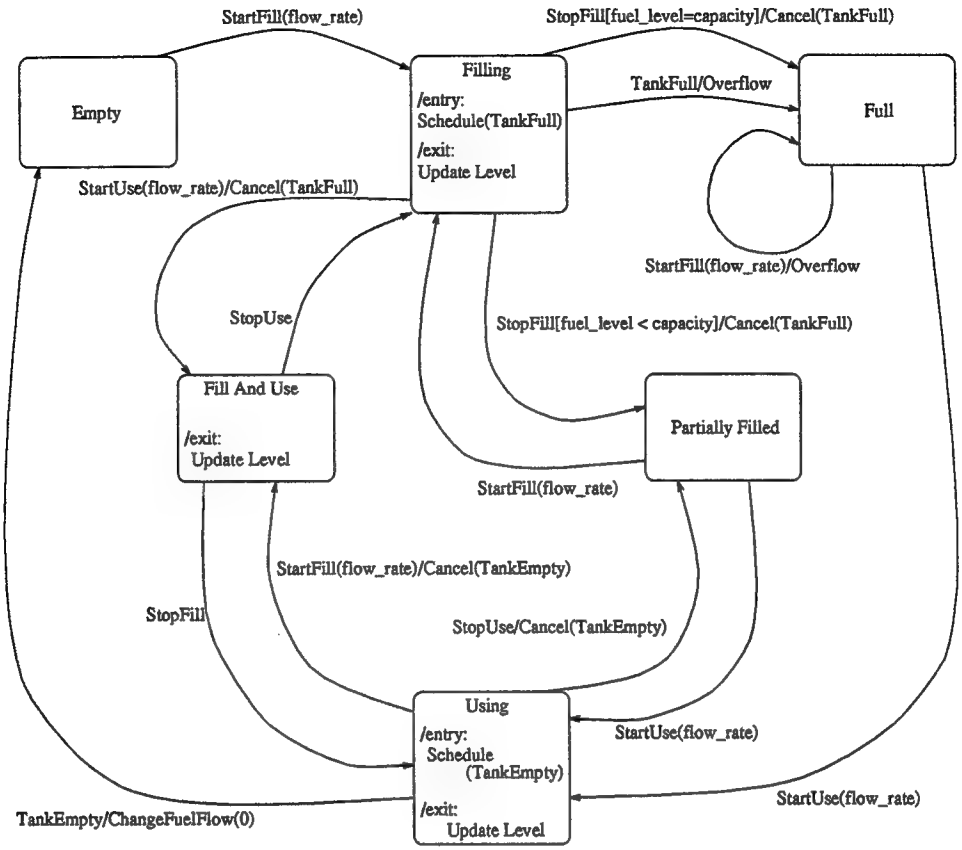

Figure 10: Fuel Tank State Transition Diagram.

PartiallyFilled State In this state the tank has some fuel but is not full. It represents a stable condition, with no input or output flow of fuel. There are no actions or activities in this state.

$$
\begin{array}{|l}
\hline
\textit{PartiallyFilled} \\
t : \textit{FuelTank} \\
\hline
t.\textit{fuel\_level} > 0 \\
t.\textit{fuel\_level} < t.\textit{capacity} \\
t.\textit{input\_flow\_rate} = 0 \\
t.\textit{output\_flow\_rate} = 0 \\
\hline
\end{array}
$$

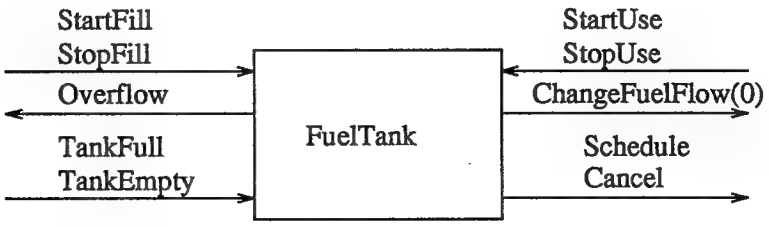

Figure 11: Fuel Tank Event Flow Diagram.

Table 3: Fuel Tank State Transition Table.

Current	Event	Parameters	Guard	Next	Action
Empty	StartFill	flow_rate		Filling	
Filling	StopFill		$fuel\_level = capacity$	Full	Cancel(TankFull)
Filling	StopFill		$fuel\_level < capacity$	PartiallyFilled	Cancel(TankFull)
Filling	TankFull			Full	Overflow
Filling	StartUse	flow_rate		FillAndUse	Cancel(TankFull)
Full	StartFill	flow_rate		Full	Overflow
Full	StartUse	flow_rate		Using	
Using	TankEmpty			Empty	ChangeFuelFlow(0)
Using	StopUse			PartiallyFilled	Cancel(TankEmpty)
Using	StartFill	flow_rate		FillAndUse	Cancel(TankEmpty)
PartiallyFilled	StartFill	flow_rate		Filling	
PartiallyFilled	StartUse	flow_rate		Using	
FillAndUse	StopUse			Filling	
FillAndUse	StopFill			Using	

Full State In this state the tank is filled to capacity with fuel. It represents a stable condition, with no input or output flow of fuel. There are no actions or activities in this state.

Full

$t : FuelTank$

$t.fuel\_level = t.capacity$
$t.input\_flow\_rate = 0$
$t.output\_flow\_rate = 0$

Filling State In this state the tank is being filled. There is no output flow of fuel. Upon entering this state the fuel tank determines when it will be full, and schedules a TankFull event for that time. When leaving this state the fuel tank updates its fuel level; if an overflow hasn't occurred it cancels the scheduled TankFull event. There are no other actions or activities while in this state.

Filling

$t : FuelTank$

$t.fuel\_level \geq 0$
$t.fuel\_level \leq t.capacity$
$t.input\_flow\_rate > 0$
$t.output\_flow\_rate = 0$

Using State In this state fuel is being drawn out of the tank. There is no input flow of fuel. Upon entering this state the fuel tank determines when it will be empty, and schedules a TankEmpty event for that time. When leaving this state the fuel tank updates its fuel level; if a TankEmpty event hasn't occurred it cancels the scheduled TankEmpty event. There are no other actions or activities while in this state.

Using

$t : FuelTank$

$t.fuel\_level \geq 0$
$t.fuel\_level \leq t.capacity$
$t.input\_flow\_rate = 0$
$t.output\_flow\_rate > 0$

FillAndUse State In this state fuel is being drawn out of the tank and pumped into the tank simultaneously. In this model it is assumed that an TankFull or TankEmpty event will not occur in this state. When leaving this state the fuel tank updates its fuel level. There are no other actions or activities while in this state.

FillAndUse

$t : FuelTank$

$t.fuel\_level \geq 0$
$t.fuel\_level \leq t.capacity$
$t.input\_flow\_rate > 0$
$t.output\_flow\_rate > 0$

5.3.3 The Functional Model

The functional model for the fuel tank was developed by analyzing the actions and activities in the dynamic model. The actions consist only of sending events *Cancel* (to the NEQ), *Overflow* (to the fueling source), and *ChangeFuelFlow(0)* (to the fuel consuming object). These are trivial actions which do not require a functional model. States *Empty*, *Partially Filled*, and *Full* are idle states with no activities. Thus the top level data flow diagram, shown in Figure 12 consists of processes *Fill Tank*, *Fill and Use*, and *Use Fuel* corresponding to the activities in states *Filling*, *Fill And Use*, and *Using* respectively. In addition, the derived variable *fuel_tank_weight*'s calculation is shown as a fourth process. These four processes are further decomposed by the data flow diagrams in Figures 13 through 16. These are discussed and the dynamic schemas presented in the following sections.

Fill Tank As shown in Figure 13, process *Fill Tank* is decomposed into three leaf processes, *Determine Interval*, *Calculate Filled Level*, and *Predict Tank Full Time*.

Determine Interval In order to determine how much fuel has been added since the tank level was last updated it is necessary to determine how much simulation time has passed since the object's state was last updated. This process determines the (simulation) time increment between the current simulation time (from the simulation clock) and the time that the attribute values were valid.

DetermineInterval

$\Xi FuelTank$
$\Xi SimClock$
$interval! : SIMTIME$

$interval! = sim\_time - tank\_sim\_time$

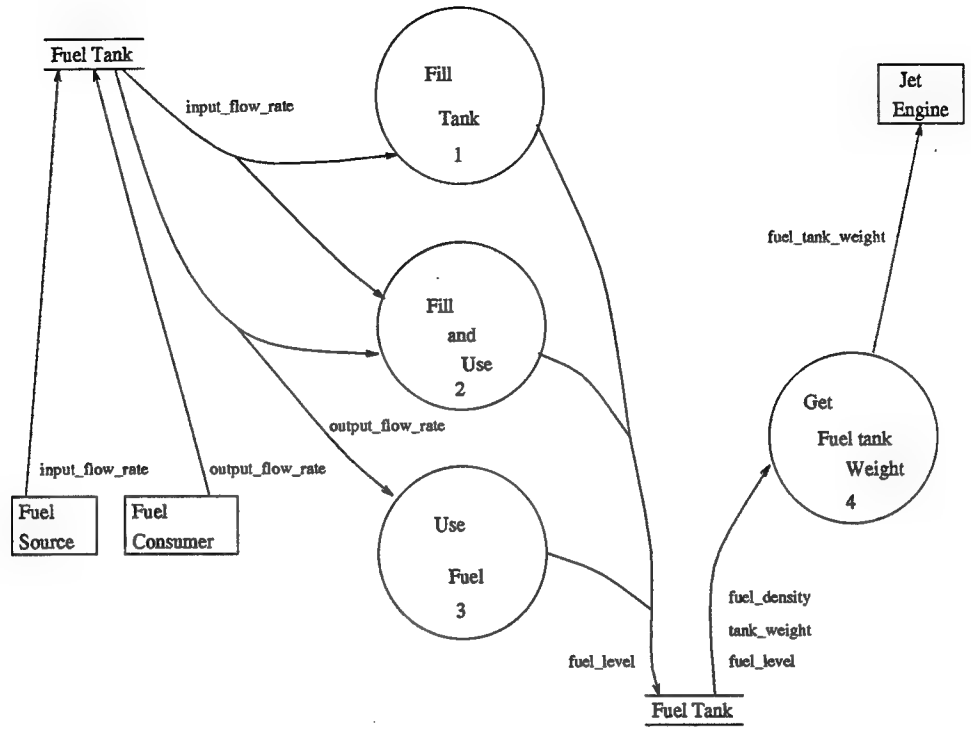

Figure 12: Fuel Tank Level 0 DFD.

Calculate Filled Level This process calculates the new level to which the tank has been filled, based on the input flow rate and the interval over which this flow rate has been sustained. This models the exit action *Update Level* for the *Filling* state.

$$
\begin{array}{|l}
\hline
\textit{CalculateFilledLevel} \\
\Delta \textit{FuelTank} \\
\Xi \textit{SimClock} \\
\textit{interval?} : \textit{SIMTIME} \\
\hline
\textit{fuel\_level}' = \textit{fuel\_level} + (\textit{interval?})(\textit{input\_flow\_rate}) \\
\textit{tank\_sim\_time}' = \textit{sim\_time} \\
\hline
\end{array}
$$

Predict Tank Full Time This process is needed when the *Filling* state is first entered. It calculates the simulation time at which the tank will overflow if still in this state, so that a *TankFull* simulation event can be scheduled.

$$
\begin{array}{|l}
\hline
\textit{PredictTankFullTime} \\
\Xi \textit{FuelTank} \\
\textit{overflow\_event\_time!} : \textit{SIMTIME} \\
\hline
\textit{overflow\_event\_time!} = \textit{tank\_sim\_time} + (\textit{capacity} - \textit{fuel\_level})/\textit{input\_flow\_rate} \\
\hline
\end{array}
$$

Fill and Use As shown in Figure 14, process *Fill and Use* is decomposed into three leaf processes, *Determine Interval*, *Calculate Net Flow*, and *Calculate Fill-Use Level*. Process *Determine Interval* is identical to that in Figure 13 and is "reused" here.

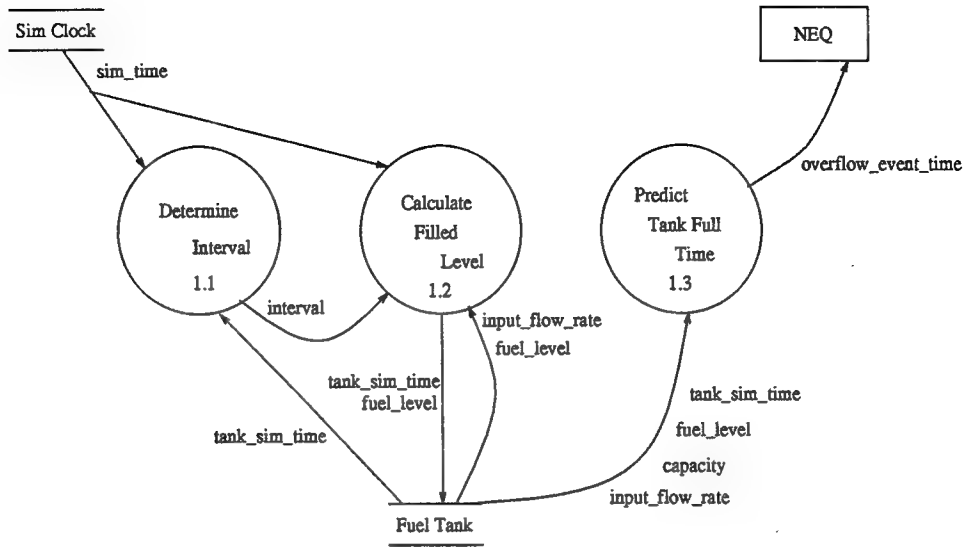

Figure 13: Fuel Tank DFD for Fill Tank.

Calculate Net Flow This process calculates the net flow rate into the fuel tank due to simultaneous filling and using.

$$
\begin{array}{|l}
\hline
\textit{CalculateNetFlow} \\
\Xi \textit{FuelTank} \\
\textit{net\_flow\_rate}! : \mathcal{R} \\
\hline
\textit{net\_flow\_rate}! = \textit{input\_flow\_rate} - \textit{output\_flow\_rate} \\
\hline
\end{array}
$$

Calculate Fill-Use Level This process calculates the new level to which the tank has been filled, based on the net flow rate and the interval over which this flow rate has been sustained. This models the exit action *Update Level* for the *Fill And Use* state.

$$
\begin{array}{|l}
\hline
\textit{CalculateFill} - \textit{UseLevel} \\
\Delta \textit{FuelTank} \\
\Xi \textit{SimClock} \\
\textit{net\_flow\_rate}? : \mathcal{R} \\
\textit{interval}? : \textit{SIMTIME} \\
\hline
\textit{fuel\_level}' = \textit{fuel\_level} + (\textit{interval}?)(\textit{net\_flow\_rate}?) \\
\textit{tank\_sim\_time}' = \textit{sim\_time} \\
\hline
\end{array}
$$

Use Fuel As shown in Figure 15, process *Use Fuel* is decomposed into three leaf processes, *Determine Interval*, *Calculate Used Level*, and *Predict Tank Empty Time*. Process *Determine Interval* is identical to that in Figure 13 and is "reused" here.

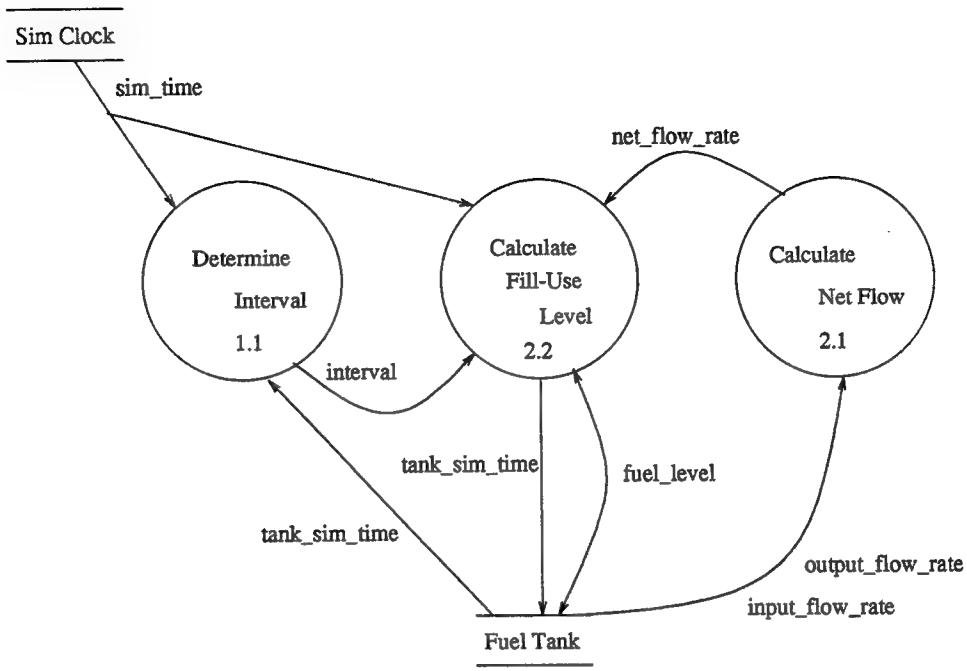

Figure 14: Fuel Tank DFD for Fill And Use.

Calculate Used Level This process calculates the new level for the tank, based on the output flow rate and the interval over which this flow rate has been sustained. This models the exit action *Update Level* for the *Using* state.

$$
\begin{array}{|l}
\hline
\textit{CalculateUsedLevel} \\
\Delta \textit{FuelTank} \\
\Xi \textit{SimClock} \\
\textit{interval?} : \textit{SIMTIME} \\
\hline
\textit{fuel\_level}' = \textit{fuel\_level} - (\textit{interval?})(\textit{output\_flow\_rate}) \\
\textit{tank\_sim\_time}' = \textit{sim\_time} \\
\hline
\end{array}
$$

Predict Tank Empty Time This process is needed when the *Using* state is first entered. It calculates the simulation time at which the tank will be empty if still in this state, so that a *TankEmpty* simulation event can be scheduled.

$$
\begin{array}{|l}
\hline
\textit{PredictTankEmptyTime} \\
\Xi \textit{FuelTank} \\
\textit{tank\_empty\_event\_time!} : \textit{SIMTIME} \\
\hline
\textit{tank\_empty\_event\_time!} = \textit{tank\_sim\_time} + \textit{fuel\_level}/\textit{output\_flow\_rate} \\
\hline
\end{array}
$$

Get Fuel Tank Weight As shown in Figure 16, process *Get Fuel Tank Weight* is decomposed into two leaf processes, *Determine Fuel Weight* and *Calculate Total Weight.* While this might be considered as decomposing a process too far due to the simplicity of the calculation involved, it is included for illustrative purposes.

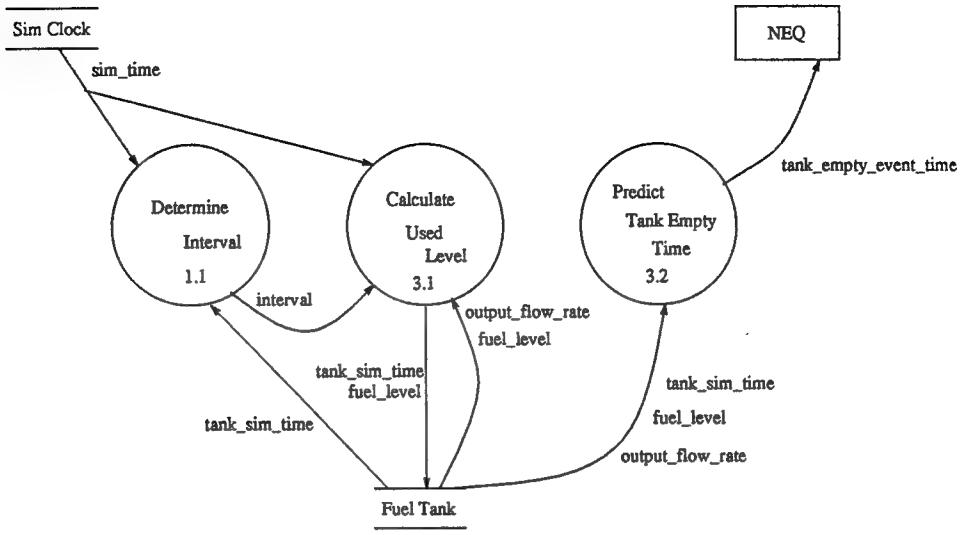

Figure 15: Fuel Tank DFD for Use Fuel.

Determine Fuel Weight This process calculates the weight of the current fuel load.

$$
\begin{array}{|l}
\hline
\textit{DetermineFuelWeight} \\
\Xi \textit{FuelTank} \\
\textit{fuel\_weight}! : \mathcal{R} \\
\hline
\textit{fuel\_weight}! = (\textit{fuel\_level})(\textit{fuel\_density}) \\
\hline
\end{array}
$$

Calculate Total Weight This process calculates the total weight of the tank and fuel.

$$
\begin{array}{|l}
\hline
\textit{CalculateTotalWeight} \\
\Xi \textit{FuelTank} \\
\textit{fuel\_weight}? : \mathcal{R} \\
\textit{fuel\_tank\_weight}! : \mathcal{R} \\
\hline
\textit{fuel\_tank\_weight}! = \textit{fuel\_weight}? + \textit{tank\_weight} \\
\hline
\end{array}
$$

5.4 Jet Engine

This is a simple engine model that converts an input fuel flow rate to a proportional output thrust. This model does not include physical dimensions. Static attributes include the manufacturer and model number, weight, maximum input fuel flow rate, and a constant thrust factor. State variables include the current input fuel flow rate and current thrust (a derived attribute). Constant thrust is generated proportional to the input fuel flow rate.

The dynamic behavior consists of states OFF and RUNNING. The engine is OFF whenever the input fuel flow rate is zero. (In a real jet engine the fuel flow is started and then an ignition spark (event) is provided to start the engine. A flameout can cause the engine to stop with fuel still flowing. Such behavior is not modeled here.) The engine responds to messages to start and to change the input fuel flow rate, and sends changes in thrust to any connected vehicle.

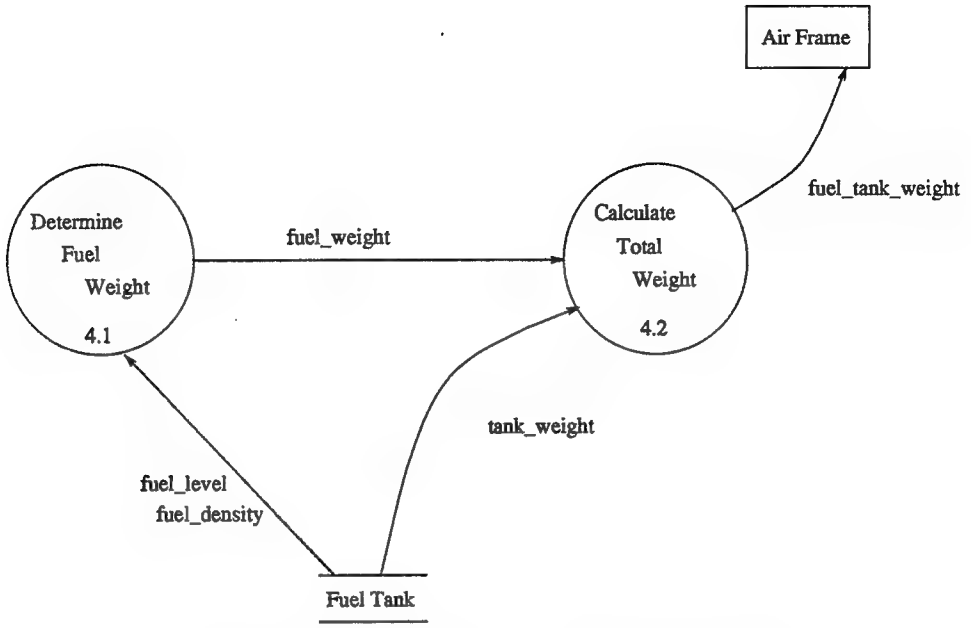

Figure 16: Fuel Tank DFD for Get Fuel Tank Weight.

5.4.1 The Object Model

The object model consists of the single object *Jet Engine*, since it has neither components nor a superclass. The *Z* Static Schema is as follows.

Let **MODEL_TYPE** be the set of all jet engine model numbers.

$[MODEL\_TYPE]$

$$
\begin{array}{|l}
\hline
JetEngine \\
\hline
manufacturer : \mathrm{seq}\, CHAR \\
model\_num : MODEL\_TYPE \\
engine\_weight : \mathcal{R} \\
maximum\_fuel\_flow\_rate : \mathcal{R} \\
thrust\_factor : \mathcal{R} \\
current\_fuel\_flow\_rate : \mathcal{R} \\
current\_thrust : \mathcal{R} \\
\hline
engine\_weight > 0 \\
maximum\_fuel\_flow\_rate > 0 \\
thrust\_factor > 0 \\
current\_thrust \geq 0 \\
current\_fuel\_flow\_rate \geq 0 \\
current\_fuel\_flow\_rate \leq maximum\_fuel\_flow\_rate \\
current\_thrust = (thrust\_factor)(current\_fuel\_flow\_rate) \\
\hline
\end{array}
$$

5.4.2 The Dynamic Model

The dynamic model state transition diagram is shown in Figure 17, and the corresponding state transition table is in Table 4. The partial event flow diagram is shown in Figure 18.

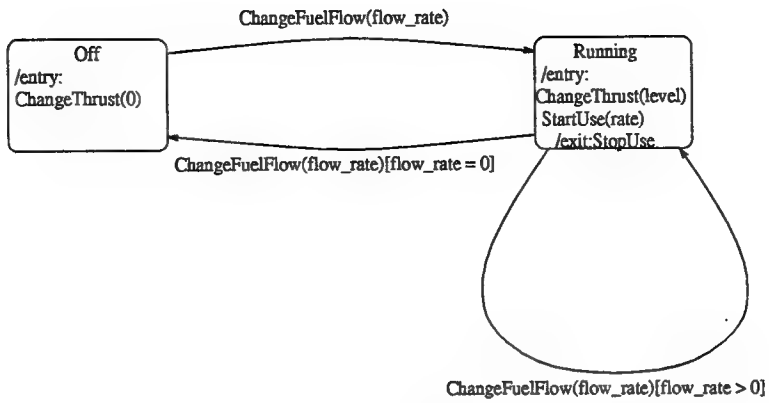

Figure 17: Jet Engine State Transition Diagram.

Table 4: Jet Engine State Transition Table.

Current	Event	Parameters	Guard	Next	Action
Off	ChangeFuelFlow	flow_rate		Running	
Running	ChangeFuelFlow	flow_rate	flow_rate > 0	Running	
Running	ChangeFuelFlow	flow_rate	flow_rate = 0	Off	

Off State In this state the engine is not running since the input flow rate is zero. Upon entering this state the engine generates a ChangeThrust(0) event. There are no other actions or activities while in this state.

Off

$e : JetEngine$

$e.current\_fuel\_flow\_rate = 0$

Running State In this state the engine is running since the input flow rate is greater than zero. Upon entering this state the engine generates a ChangeThrust(thrust_level) event, where the thrust level depends on the input fuel flow rate, and a StartUse(rate) event to the fuel source (Fuel Tank). On exit a StopUse event is sent to the fuel source (Fuel Tank). There are no other actions or activities while in this state.

Running

$e : JetEngine$

$e.current\_fuel\_flow\_rate > 0$

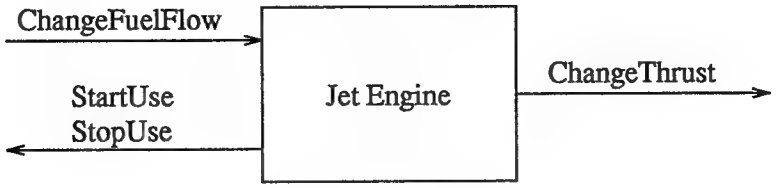

Figure 18: Jet Engine Event Flow Diagram.

5.4.3 The Functional Model

The data flow diagram for the jet engine is shown in Figure 19. This process is simple enough that it is

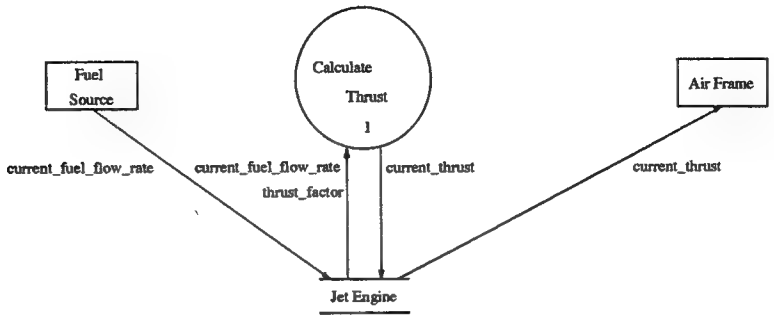

Figure 19: Jet Engine DFD for Calculate Thrust.

decomposed no further. Process **Calculate Thrust** calculates the thrust for a given input fuel flow rate.

$$
\begin{array}{|l}
\hline
\textit{CalcThrust} \\
\Delta \textit{JetEngine} \\
\hline
\textit{current\_thrust}' = (\textit{thrust\_factor})(\textit{current\_fuel\_flow\_rate}) \\
\hline
\end{array}
$$

5.5 Airframe

This is a model of an airborne vehicle. It represents a moving object in a three dimensional coordinate system. Location and movement of a point mass is represented.

Static attributes include serial number and manufacturer, weight, and two colors. State variables include the location (x, y, and z), velocity, and acceleration of the center of mass, the azimuth and elevation of the centerline, the speed (a derived attribute), and the simulation time at which these values are (were) valid.

The dynamic behavior modeled includes launching the airframe into powered flight, in which the effects of gravity are ignored, until the fuel runs out, then allowing the airframe to follow an inertial trajectory until it crashes into the ground.

5.5.1 The Object Model

The object model consists of the single object *Airframe*, since it has neither components nor a superclass. The *Z* Static Schema is as follows.

$[AF\_MODELS]$

Airframe

$tail\_num : \text{seq}\, ALPHANUM$
$model\_type : AF\_MODELS$
$airframe\_weight : \mathcal{R}$
$attached\_weight : \mathcal{R}$
$airframe\_simtime : SIMTIME$
$applied\_thrust : \mathcal{R}$
$X : \mathcal{R}$
$Y : \mathcal{R}$
$Z : \mathcal{R}$
$V_x : \mathcal{R}$
$V_y : \mathcal{R}$
$V_z : \mathcal{R}$
$A_x : \mathcal{R}$
$A_y : \mathcal{R}$
$A_z : \mathcal{R}$
$\theta : \mathcal{R}$
$\phi : \mathcal{R}$
$speed : \mathcal{R}$

$airframe\_weight \geq 0.0$
$attached\_weight \geq 0.0$
$applied\_thrust \geq 0.0$
$(\theta \geq -180.0) \wedge (\theta \leq +180.0)$
$(\phi \geq 0.0) \wedge (\phi \leq 90.0)$
$speed = \sqrt{V_x^2 + V_y^2 + V_z^2}$

5.5.2 The Dynamic Model

The dynamic model state transition diagram is shown in Figure 20, and the corresponding state transition table is in Table 5. The partial event flow diagram is shown in Figure 21.

Table 5: Airframe State Transition Table.

Current	Event	Parameters	Guard	Next	Action
Ready	ChangeThrust	value	value > 0	PoweredFlight	
PoweredFlight	ChangeThrust	value	value = 0	InertialFlight	
InertialFlight	ChangeThrust	value	value > 0	PoweredFlight	Cancel(Collision)
InertialFlight	Collision			Crashed	

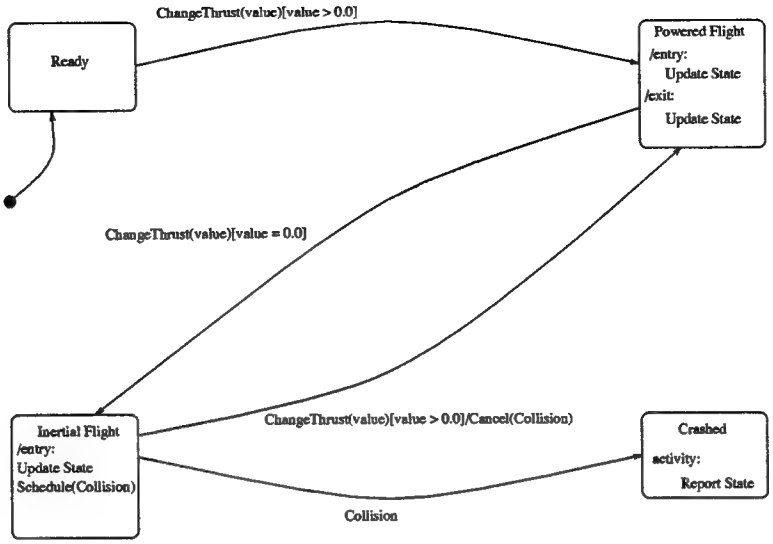

Figure 20: Airframe State Transition Diagram.

Figure 21: Airframe Event Flow Diagram.

Ready State In this state the airframe has been aimed, and is ready to launch. The applied thrust is zero. There are no actions or activities while in this state.

Ready

$a : Airframe$

$a.applied\_thrust = 0.0$
$a.V_x = 0.0$
$a.V_y = 0.0$
$a.V_z = 0.0$
$a.A_x = 0.0$
$a.A_y = 0.0$
$a.A_z = 0.0$

Powered Flight State In this state thrust is applied. On entering this state the airframe updates its simulation time and acceleration. When leaving this state the airframe updates its position and velocity. There are no other actions or activities while in this state.

PoweredFlight

$a : Airframe$

$a.applied\_thrust > 0.0$

Inertial Flight State In this state there is no applied thrust. On entering this state the airframe updates its simulation time and acceleration. It then determines when it will hit the ground and schedules a collision event. When leaving this state the airframe updates its position and velocity. If a collision hasn't occurred it cancels the previously scheduled collision event. There are no other actions or activities while in this state.

InertialFlight

$a : Airframe$

$a.applied\_thrust = 0.0$
$(a.V_x > 0.0) \vee (a.V_y > 0.0) \vee (a.V_z > 0.0)$

Crashed State In this state the airframe has hit the ground. There are no actions or activities while in this state.

Crashed

$a : Airframe$

$a.applied\_thrust = 0.0$
$a.V_x = 0.0$
$a.V_y = 0.0$
$a.V_z = 0.0$
$a.A_x = 0.0$
$a.A_y = 0.0$
$a.A_z = 0.0$

5.5.3 The Functional Model

The functional model is based on the state transition diagram. Since nothing happens in either the ready or the crashed states, there are two main processes to be performed, as indicated in the top level DFD of Figure 22, corresponding to the states *Powered Flight* and *Inertial Flight.*

Perform Powered Flight As indicated in Figure 23, process *Perform Powered Flight* is further decomposed into *Determine Interval*, *Calculate Position*, *Calculate Velocity*, and *Calculate Acceleration.*

Determine Interval In order to determine how far the vehicle has traveled since the last update, it is necessary to determine how much simulation time has passed since the object's state was last updated. This process determines the (simulation) time increment between the current simulation time (from the simulation clock) and the time that the attribute values were valid.

DetermineInterval

$\Xi Airframe$
$\Xi SimClock$
$interval! : SIMTIME$

$interval! = sim\_time - airframe\_simtime$

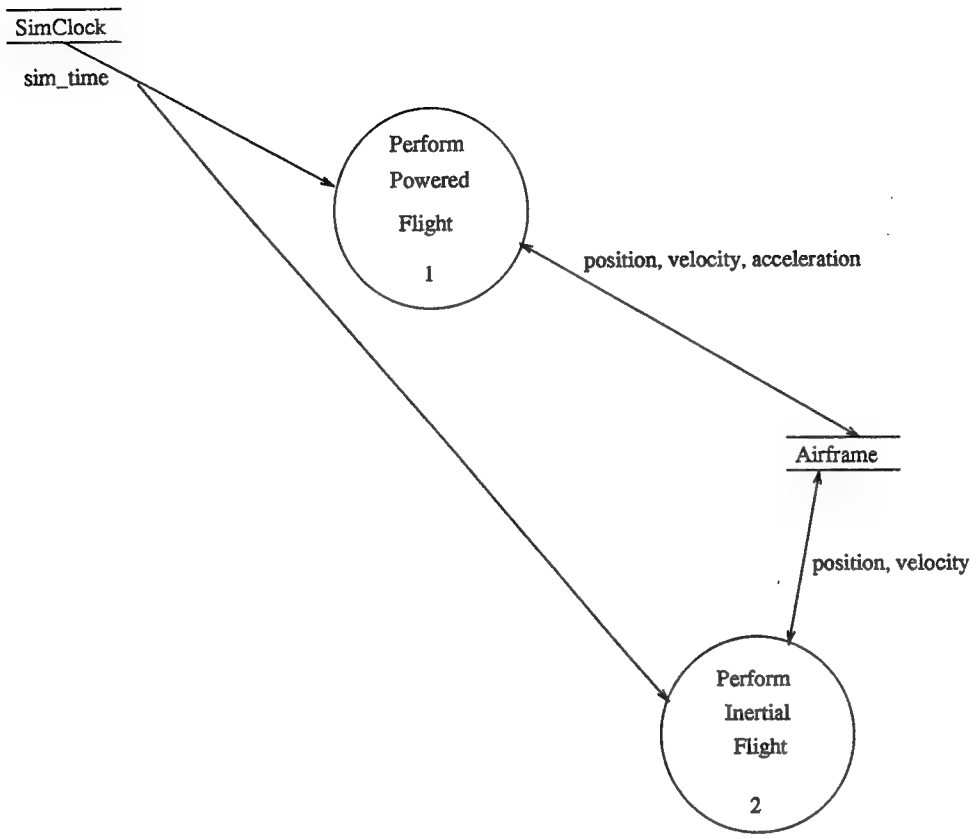

Figure 22: Airframe Level 0 DFD.

Calculate Position This process calculates the new position of the vehicle based on the interval traveled at constant acceleration. Effects of gravity and air drag are ignored.

CalculatePosition

$\Delta Airframe$
$interval? : SIMTIME$

$X' = X + V_x(interval?) + \frac{1}{2}A_x(interval?)^2$
$Y' = Y + V_y(interval?) + \frac{1}{2}A_y(interval?)^2$
$Z' = Z + V_z(interval?) + \frac{1}{2}A_z(interval?)^2$

Calculate Velocity This process calculates the new velocity of the vehicle based on constant acceleration. Effects of gravity and air drag are ignored.

CalculateVelocity

$\Delta Airframe$
$interval? : SIMTIME$

$V_x' = V_x + A_x(interval?)$
$V_y' = V_y + A_y(interval?)$
$V_z' = V_z + A_z(interval?)$

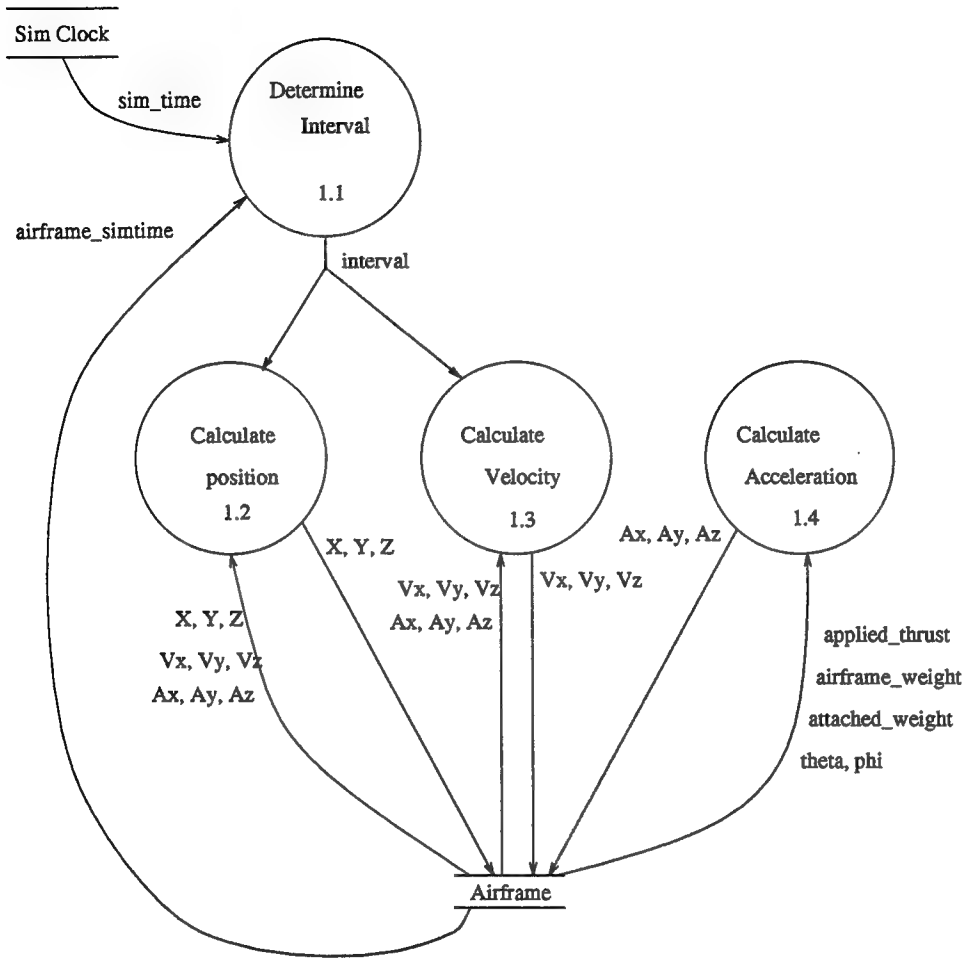

Figure 23: Airframe DFD for Perform Powered Flight.

Calculate Acceleration This process calculates the new acceleration of the vehicle based on the applied thrust, the total weight of the vehicle, and the direction of flight. Effects of gravity and air drag are ignored. The direction of flight is determined from θ and ϕ as shown in Figure 24.

CalculateAcceleration

$\Delta Airframe$

$$A'_x = \left(\frac{applied\_thrust}{airframe\_weight+attached\_weight}\right)\cos\phi\cos\theta$$
$$A'_y = \left(\frac{applied\_thrust}{airframe\_weight+attached\_weight}\right)\cos\phi\sin\theta$$
$$A'_z = \left(\frac{applied\_thrust}{airframe\_weight+attached\_weight}\right)\sin\phi$$

Perform Inertial Flight As indicated in Figure 25, process *Perform Inertial Flight* is further decomposed into *Determine Interval*, *Calculate Inertial Position*, and *Calculate Inertial Velocity*. Process *Determine Interval* is the same as in Figure 23.

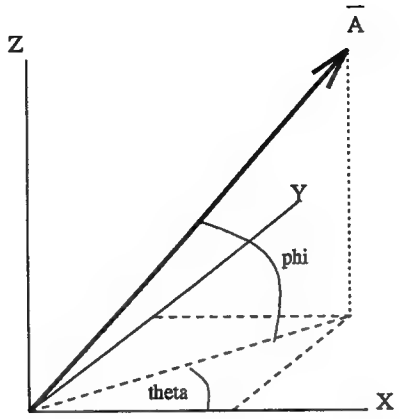

Figure 24: Airframe Azimuth and Elevation.

Calculate Inertial Position This process calculates the new position of the vehicle during inertial flight. Constant velocity in X and Y are assumed. Effects of air drag are ignored.

CalculateInertialPosition

$\Delta Airframe$
$interval? : SIMTIME$
$gravity : \mathcal{R}$

$gravity = 32.0$
$X' = X + V_x(interval?)$
$Y' = Y + V_y(interval?)$
$Z' = Z + V_z(interval?) - \frac{1}{2}(gravity)(interval?)^2$

Calculate Inertial Velocity This process calculates the new velocity of the vehicle during inertial flight. Constant velocity in X and Y are assumed. Effects of air drag are ignored.

CalculateInertialVelocity

$\Delta Airframe$
$interval? : SIMTIME$
$gravity : \mathcal{R}$

$gravity = 32.0$
$V_z' = V_z + (gravity)(interval?)$

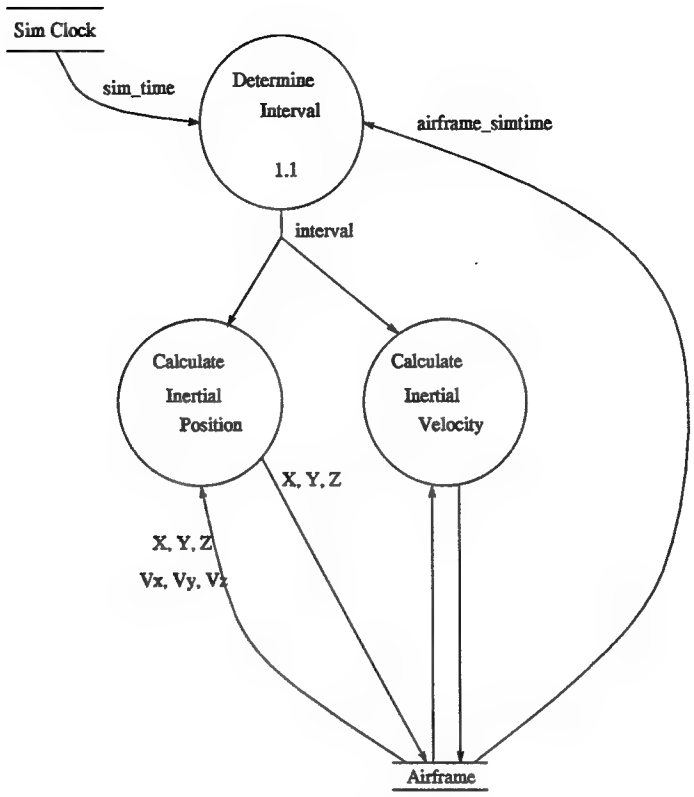

Figure 25: Airframe DFD for Perform Inertial Flight.

5.6 Rocket

As shown in Figure 9, the *Rocket* class is an aggregate with an *Airframe*, two *Fuel Tanks* and two *Jet Engine* subcomponents. Each fuel tank supplies one of the engines. The rocket is powered by the two engines. It is assumed that the two engines are mounted symmetrically on two sides of the rocket, and that they produce equal thrusts at all times. Thus the rocket is powered by a thrust equal to the sum of the thrusts of the two engines, applied along the centerline of the rocket body. The total vehicle weight is equal to that of the vehicle body, the two empty tanks, and the two engines. Fuel weight is ignored in this example.

5.6.1 The Object Model

The object model for the *Rocket* contains two each *Fuel Tank* and *Jet Engine* objects, along with one *Airframe* and an embedded association *feeds*. The subcomponents *Fuel Tank* and *Jet Engine* are modeled as sets with cardinality of two, and the *Airframe* as an attribute. (Alternative approaches would be to declare two explicit tank attributes and two explicit engine attributes, or to define a set of airframes with a cardinality of one). This is a concrete aggregate model, and includes the attributes tail_num (the rocket's tail number) and model_type. The *Z* static schema is as follows.

Let **ROCKET_MODELS** be the set of rocket models.

$[ROCKET\_MODELS]$

Rocket

$$
\begin{array}{l}
tail\_num : \text{seq}\, ALPHANUM \\
model\_type : ROCKET\_MODELS \\
frame : Airframe \\
tanks : \mathcal{P} FuelTank \\
jets : \mathcal{P} JetEngine \\
feeds : FuelTank \rightarrowtail\!\!\!\!\!\!\rightarrow JetEngine \\
\hline
\#tanks = 2 \\
\#jets = 2 \\
\text{dom}\, feeds = tanks \\
\text{ran}\, feeds = jets \\
\forall t : tanks \bullet \exists j : jets \mid ((t \mapsto j) \in feeds \\
\land\ t.fuel\_level = 0.0 \Rightarrow j.current\_fuel\_flow\_rate = 0 \\
\land\ t.fuel\_level > 0.0 \Rightarrow j.current\_fuel\_flow\_rate = j.maximum\_fuel\_flow\_rate \\
\land\ t.output\_flow\_rate = j.current\_fuel\_flow\_rate) \\
frame.attached\_weight = sum\{j : JetEngine \mid j \in jets \bullet j.engine\_weight\} \\
+sum\{t : FuelTank \mid t \in tanks \bullet t.fuel\_tank\_weight\} \\
frame.applied\_thrust = sum\{j : JetEngine \mid j \in jets \bullet j.current\_thrust\}
\end{array}
$$

5.6.2 The Dynamic Model

The state of the rocket is made up of the concurrent states of its constituent parts. However, the parts are not free to assume their states independently due to constraints between them in the context of the rocket. Table 6 lists all of the combinations of states of the airframe, the jet engine, and the fuel tank, and indicates which are possible in the rocket configuration. Note that the *Filling* and the *Fill and Use* states of the fuel tank are omitted since they are not used in this example. Since filling the tank is not being modeled, it is further assumed that the tank is initially full.

Table 7 lists all of the *allowable* combinations of states along with an appropriate title. These represent the states of the Rocket. In this case the states of the rocket coincide with the states of the airframe since there is only one combination of states of the fuel tank and engine for each airframe state. Analyzing the *cause* of each state allows the events to be derived for the rocket, resulting in the state transition diagram of Figure 26 and the corresponding state transition table in Table 8. Thus we have derived the behavior of the aggregate rocket from the constrained behaviors of its component parts. The event flow diagram is shown in Figure 27.

5.6.3 The Functional Model

In this example there are no additional calculations needed beyond those of the rocket's individual parts. Thus the rocket's functional model is the combination of all of its components' functional models.

5.7 Summary

Although this is a very simple rocket model, it illustrates the application of the *Z* extensions to Rumbaugh's informal modeling approach to develop a formal specification of a complex system by combining the formal specifications of its constituent parts.

Table 6: Available state combinations of components

Airframe	Fuel Tank	Engine	Rocket	Reason
Ready	Empty	Off	Not Possible	Tank initially full
Ready	Empty	Running	Not possible	Empty and Running
Ready	PartiallyFilled	Off	Not Possible	Tank initially full
Ready	PartiallyFilled	Running	Not possible	Running and not Using
Ready	Full	Off	Possible	
Ready	Full	Running	Not possible	Running and not Using
Ready	Using	Off	Not possible	Using and Off
Ready	Using	Running	Not possible	Ready and Running
PoweredFlight	Empty	Off	Not possible	Off and PoweredFlight
PoweredFlight	Empty	Running	Not possible	Running and not Using
PoweredFlight	PartiallyFilled	Off	Not possible	PoweredFlight and Off
PoweredFlight	PartiallyFilled	Running	Not possible	Running and not Using
PoweredFlight	Full	Off	Not possible	PoweredFlight and Off
PoweredFlight	Full	Running	Not possible	Running and not Using
PoweredFlight	Using	Off	Not possible	PoweredFlight and Off
PoweredFlight	Using	Running	Possible	
InertialFlight	Empty	Off	Possible	
InertialFlight	Empty	Running	Not possible	InertialFlight and Running
InertialFlight	PartiallyFilled	Off	Not possible	PartiallyFilled and Off
InertialFlight	PartiallyFilled	Running	Not possible	Running and not Using
InertialFlight	Full	Off	Not possible	Full after launch
InertialFlight	Full	Running	Not possible	Running and not Using
InertialFlight	Using	Off	Not possible	Using and Off
InertialFlight	Using	Running	Not possible	InertialFlight and Running
Crashed	Empty	Off	Possible	
Crashed	Empty	Running	Not possible	Empty and Running
Crashed	PartiallyFilled	Off	Not possible	Off implies Empty
Crashed	PartiallyFilled	Running	Not possible	Running and not Using
Crashed	Full	Off	Not possible	Crashed and Full
Crashed	Full	Running	Not possible	Running and not Using
Crashed	Using	Off	Not possible	Using and Off
Crashed	Using	Running	Not possible	Crashed and Running

6 Summary

The approach described in this report of integrating informal methods with formal methods has been taught in an introductory course in Software Engineering at the Air Force Institute of Technology. In the first two course offerings, we taught informal object-oriented modeling for the first seven weeks of the quarter, then introduced formal methods during the last three, loosely coupling the two using objects as the basis for the *Z* schemas. Students tended to treat this as a change in topic, and found the three week formal portion somewhat difficult to understand as a stand-alone topic. By introducing the mathematics and *Z* formalism at the beginning of the quarter, there is more time to "relearn" the mathematics over the space of the quarter, and integrating the formalism as a way to document the informal model in the data dictionary adds to the sense of purpose of the formalism.

However, some of the students still don't see the usefulness of the formal extensions. They express the perception that unnecessary complexity and difficulty has been added to an existing methodology. Part of this is due to the lack of an executable formal language and a general lack of automated tools for *Z* that build on the formal specification. Much of the work in performing *Z* data and operation refinement as well

Table 7: Available states of the Rocket

Airframe	Fuel Tank	Engine	Rocket
Ready	Full	Off	Pre-Flight
PoweredFlight	Using	Running	Accelerating
InertialFlight	Empty	Off	Coasting
Crashed	Empty	Off	Impacted

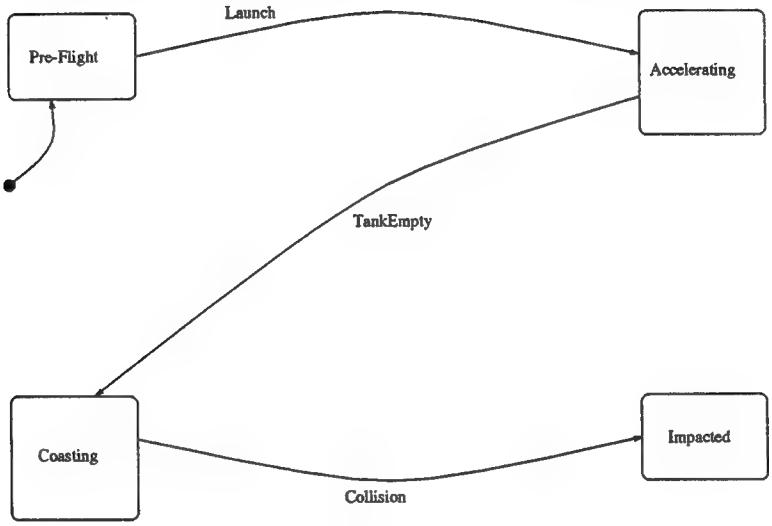

Figure 26: Rocket State Transition Diagram.

as the final mapping to some implementation language must be done manually. Another problem is cultural in nature. While it is improving with each new class, many of our students have seen nothing concerning the use of formal methods other than possibly proof of correctness techniques (which provided an extremely negative impression).

This course is a required course for all graduate computer students (approximately 50 per year). Therefore, we have exposed a substantial number of students to the use of formal methods for system development. In addition to this course, all software engineering students (approximately 15 per year) are required to take an additional course entitled *Formal-Based Methods in Software Engineering.* This course uses the Software Refinery environment which contains a mathematically-based, wide-spectrum formal specification language called Refine which is also executable. Refine integrates set theory, logic, objects, a formal object base, transformation rules, and pattern matching [6]. Additionally, the Software Refinery environment includes a parser generator and X11 interface toolkit. Essentially, Software Refinery can be used to build program transformation systems [15] in which high level formal specifications are used as the basis for synthesizing lower level implementations in languages like C and Ada. With the basis provided by the introductory

Table 8: Rocket State Transition Table.

Current	Event	Parameters	Guard	Next	Action
Pre-Flight	Launch			Accelerating	
Accelerating	TankEmpty			Coasting	
Coasting	Collision			Impacted	

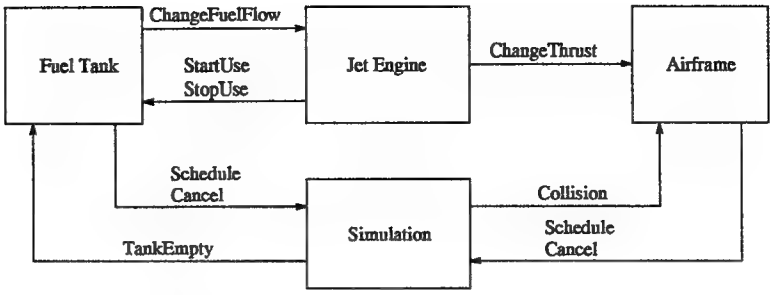

Figure 27: Rocket Event Flow Diagram.

course, the software engineering students are in a position to build object model transformation systems. In particular, it is shown how object-based domain models can be used to develop domain-specific languages and how these languages can be transformed in a behavior-preserving way using object model to object model transformations. Thus, the elements of how to build a specification to code level transformation system are not only discussed in class, but practical experience with constructing and using such systems is obtained.

Overall, this course has been very successful at simultaneously introducing students to formal methods and object-oriented modeling. This course, coupled with our advanced course in formal-based methods, where we do have more sophisticated tool support, produces a very enlightened set of software engineers.

A Math Review

This section reviews the basic discrete math used in Z, and introduces some new notation.

A.1 Set Definition

In using Z for formal specification, all variables must be defined over a set. Therefore it is important to be able to define sets clearly. This section reviews some basic methods of defining sets, and discusses issues of set membership and type declarations.

A.1.1 Basic Sets

A basic set can be defined by simply stating what its members are. No further detail is provided about the set's members.

Let $PERSONS$ be the set of all persons.
Let $STUDENTS$ be the set of all students.
Let $SSAN$ be the set of all social security numbers.
Let $PERNAMES$ be the set of all persons' first, middle or last names.
Let $CHAR$ be the set of printable characters.
Let $DIGIT$ be the set of printable digits. ($DIGIT \subset CHAR$).

A.1.2 Standard Sets

$\mathcal{Z}$ Integers.
$\mathcal{N}$ Natural numbers.
$\mathcal{N}_1$ Natural numbers greater than zero.
$\mathcal{R}$ Real numbers.

A.1.3 Explicit Enumeration

A set can be defined by listing its members in braces. The symbol "$\widehat{=}$" is used to define a set.

$Gradeable \mathrel{\widehat{=}} \{\text{Homework, Project, Midterm, Final}\}$

This syntax could also be used to define basic sets.

$STUDENTS \mathrel{\widehat{=}} \{All\ students\}$

A.1.4 Powerset

The powerset of a set S (written as $\mathcal{P}S$) is the set of all subsets of the specified set, including the null set $\{\ \}$ and the specified set itself.

$S \mathrel{\widehat{=}} \{1, 2, 3\}$

$\mathcal{P}S = \{\{\ \}, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$

A.1.5 Set Membership

The symbol $\in$ represents set membership. If variable x is a member of set A, then we write $x \in A$.

If $x =$ Midterm then $x \in Gradeable$.

Note that "$x \in Gradeable$" is a *predicate* that can be true or false (see Section A.5). Thus if $x =$ Quiz then $x \in Gradeable$ would be false.

A.1.6 Type Declaration

A variable can be declared to be of the same type as members of a set. That is, the variable is declared "over the set."

x : *Gradeable*
y : $\mathcal{N}$
s : SSAN

Thus $s : SSAN$ declares that s is a variable that is of the type of a member of set $SSAN$. Note the difference between the type declaration and set membership. If x : *Gradeable*, then $x \in$ *Gradeable* is true for any value of x.

A.1.7 Set Types

If the variable to be declared is to be a set itself, then the powerset operator can be used. For example,

exams : $\mathcal{P}$ *Gradeable*

declares *exams* to be a *set* of entities of the same type as in *Gradeable*. In set notation (Section A.4), *exams* $\subseteq$ *Gradeable*.

A.2 Cartesian Product

While all members of a set must be of the same type, a set can be defined over any type. Specifically, a set member could be an ordered pair. For example, consider the set of pairs { (a,2), (b,1), (c,2) }. Often it is useful to be able to declare a variable as an ordered pair or as a set of ordered pairs.

Given sets T and U the Cartesian Product, written $T \times U$, is the set of *all* ordered pairs whose first element is from T and whose second element is from U.

For example, if

$T \mathrel{\widehat{=}} \{a, b, c\}$

$U \mathrel{\widehat{=}} \{1, 2\}$

then,

$T \times U = \{(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)\}$

$T \times T = \{(a,a), (a,b), (a,c), (b,a), (b,b), (b,c), (c,a), (c,b), (c,c)\}$

$U \times U = \{(1,1), (1,2), (2,1), (2,2)\}$

Now let

$STUDENTS \mathrel{\widehat{=}} \{$Smith, Green$\}$

Gradeable $\mathrel{\widehat{=}} \{$Homework, Project, Midterm, Final$\}$

Then we can declare

taken : $STUDENTS \times$ *Gradeable*

That is, "taken" is a pair (a,b) where

$a \in STUDENTS$

$b \in$ *Gradeable*

Thus

$$STUDENTS \times Gradeable = \{(\text{Smith, Homework}), (\text{Smith, Midterm}), (\text{Smith, Final}), (\text{Green, Homework}), (\text{Green, Midterm}), (\text{Green, Final})\}$$

If

$$taken = (\text{Smith, Final})$$

then, since *taken* is a set *element*, we can write

$$taken \in STUDENTS \times Gradeable$$

What if we want the set of students and the gradeables they have taken at some point in time (i.e. not all students have taken all gradeables)? We can use the power set to define the set *all_taken*. (Note that *taken* is a pair, while *all_taken* is a set of pairs).

$$all\_taken : \mathcal{P}(STUDENTS \times Gradeable)$$

Thus, for example, if everyone has taken only the midterm,

$$all\_taken \mathrel{\widehat{=}} \{(\text{Smith, Midterm}), (\text{Green, Midterm})\}$$

then, since *all_taken* is a *set*, we can write (see Section A.4)

$$all\_taken \subseteq STUDENTS \times Gradeable$$

Similarly we can define $A \times B \times C$ as the set of all 3-tuples (a_i, b_j, c_k) and so forth.

A.3 Set Operators

Three set operators are of interest here. These are the *union* $\cup$, the *intersection* $\cap$, and subtraction $\setminus$.

$A \cup B$ is the set of all x such that $x \in A$ or $x \in B$ or x is in both.
$A \cap B$ is the set of all x such that $x \in A$ and $x \in B$.
$A \setminus B$ is the set of all x such that $x \in A$ and $x \notin B$.

All members of a set must be of the same type. The number of members in a set is defined as the *cardinality* of the set, designated with the "#".

$\#\{a,c,f\} = 3$

A.4 Set Comparison

Sets can be compared as follows.

$A \subseteq B$ (A is a *subset* of B) is true if all members of A are also members of B.

$A \subset B$ (A is a *proper subset* of B) is true if all members of A are also members of B, but not all members of B are in A.

$A = B$ (A equals B) is true if all members of A are also members of B, and all members of B are members of A.

$A \neq B$ (A not equal to B) is true if the members of A are not the same as the members of B.

A.5 Predicates

A *proposition* is a statement that is true or false.

"The student Smith has a GPA greater than 3.0"

"The student Jones is enrolled in CSCE 594"

A *predicate* is a proposition with a variable whose value makes the statement true or false. NOTE: For any predicate there exists a set of values (possibly the empty set) for which the predicate is true.

"The student s is enrolled in CSCE 594"

Note that a predicate can include comparators, for example $<, >, =, \subseteq$ or anything that will evaluate to true or false.

Predicates can be combined with the following operators.

$\neg$	not
$\wedge$	and
$\vee$	or
$\Rightarrow$	implication
$\Leftrightarrow$	equivalence

Remember not to treat implication as "p causes q" but rather "if p then q." Implication (as with the others in the list) is a predicate that is true or false, according to the following truth table.

p	q	$p \Rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

For example,

Student S is enrolled in CSCE 594 $\Rightarrow$ student s has taken CSCE 431.

Is this true or false? This predicate will be true for any student s who has taken CSCE 431 and is enrolled in CSCE 594. However, the predicate will *also* be true for all students s who are *not* enrolled in CSCE 594!

Precedence Sequences of the same operator are evaluated left to right.

$a \vee b \vee c = (a \vee b) \vee c$

The precedence of different operators is: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

$\neg a \wedge b = (\neg a) \wedge b$

$a \vee b \Leftrightarrow c \wedge \neg d = (a \vee b) \Leftrightarrow (c \wedge (\neg d))$

It is frequently helpful to include parentheses to help clarify the expression.

A.6 Quantification

Quantification allows us to assert a predicate that involves one of a set or all of a set. The general form of quantification is

$\exists$ *signature* • *predicate*

$\forall$ *signature* • *predicate*

where the signature is a list of data types used to formulate the predicate. That is, the signature defines the vocabulary for making mathematical statements expressed as predicates. The predicate describes relationships among the identifiers (variables) as well as constraints.

The signature consist of a set of identifier/class declarations as follows:

1. Identifier/class is a series of identifiers separated by commas followed by a colon and a class.
2. Each identifier/class is separated by semicolons.
3. Class represents a class or set of objects over which the quantification holds.

For example:

$\exists i : \mathcal{N} \bullet i > 10$
This is an example of *existential quantification.* It asserts that there is at least one value in the set of natural numbers that is greater than ten. Clearly this predicate is always true.

$\exists j : 1 \mathinner{..} 100 \bullet j \geq 3 \wedge j \leq 8 \wedge j^2 = 49$
This predicate is also true, since $j = 7$ satisfies the predicate.

$\forall i : 1 \mathinner{..} 10 \bullet i^3 \leq 1000$
This is an example of *universal quantification.* It asserts that *all* values from 1 to 10 have cubes that are less than or equal to 1000. This predicate is also true.

Let $STUDENTS$ be the set of all students.
Let $IN594$ be the set of students enrolled in CSCE 594.
Let $HAS431$ be the set of students who have taken CSCE 431.

Consider the meaning of the following quantified expressions, when true.

$\exists s : STUDENTS \bullet s \in IN594 \wedge s \in HAS431$
At least one student is enrolled in CSCE 594 and has taken CSCE 431.

$\forall s : STUDENTS \bullet s \in IN594 \wedge s \in HAS431$
All students are enrolled in CSCE 594 and have taken CSCE 431.

$\exists s : IN594 \bullet s \in HAS431$
At least one student enrolled in CSCE 594 has taken CSCE 431.

$IN594 \subseteq STUDENTS$
Everyone enrolled in CSCE 594 is a student (always true).

$\exists s : STUDENTS \bullet s \in IN594 \Rightarrow s \in HAS431$
At least one student is either enrolled in CSCE 594 and has taken CSCE 431 or is not enrolled in CSCE 594. (Not all students are enrolled in CSCE 594 without having taken CSCE 431).

$\forall s : STUDENTS \bullet s \in IN594 \Rightarrow s \in HAS431$
All students who are enrolled in CSCE 594 have taken CSCE 431. (No students are enrolled in CSCE 594 who have not taken CSCE 431).

A.7 Set Comprehension

Earlier it was pointed out that for any predicate there is a set of values (possibly empty) for which the predicate is true. *Set Comprehension* is a way of using this fact to define a set. The general "Set Former" construct is as follows.

$$\{x : S \mid P(x) \bullet t(x)\}$$

Thus, the elements of this set are all elements which have the form $t(x)$ (the *term*) such that x is a bound variable (defined by S) and satisfies the predicate $P(x)$. Note that the *term* is optional if $t(x) = x$.

Examples:

$$\{x : \mathcal{N} \mid x^3 \in Even \bullet x + 1\} = \{1, 9, 65, \ldots\}$$

That is, the set of all numbers which are one more than an even cube.

$S_1 \mathrel{\widehat=} \{n : \mathcal{N}_1 \mid n^3 > 10 \bullet n\} = \mathcal{N} \setminus \{0, 1, 2\} = \{3, 4, 5, \ldots\}$
This is the set of all natural numbers whose cube is greater than 10.

$S_2 \mathrel{\widehat=} \{n : \mathcal{N} \mid n > 10 \bullet (n, n^2)\} = \{(11, 121), (12, 144), \ldots\}$
This is the set of all pairs where the first element is a natural number greater than ten and the second element is its square.

$S_3 \mathrel{\widehat=} \{x, y : \mathcal{N} \mid x + y = 100 \bullet (x, y)\} = \{(0, 100), (1, 99), \ldots, (100, 0)\}$
This is the set of all pairs of natural numbers whose sum is 100.

Note that for S_1 and S_3 the *term* is optional.

$IN594 \mathrel{\widehat=} \{s : STUDENTS \mid s \text{ is enrolled in CSCE 594}\}$
$HAS\_PREREQ \mathrel{\widehat=} \{s : STUDENTS \mid s \in IN594 \land s \in HAS431\}$
or
$HAS\_PREREQ \mathrel{\widehat=} \{s : STUDENTS \mid s \in IN594 \cap HAS431\}$

Let SSAN be the set of all social security numbers.
$STUDENTS \mathrel{\widehat=} \{s : SSAN;\ n : PERNAMES \mid$ a person with number s and name n is a student$\}$
or
$STUDENTS \mathrel{\widehat=} \{s : SSAN;\ n : PERNAMES \mid TRUE\} = \{s : SSAN;\ n : PERNAMES\}$
or
$STUDENTS : \mathcal{P}(SSAN \times PERNAMES)$

A.8 Relations

We have introduced the concept of an ordered set of pairs (or of *tuples* in general). We now use that concept to define how elements of different sets are related. For example, consider the relationship between students and the faculty members who are their advisors.

Given that sets $STUDENTS$ and $FACULTY$ are defined, then

$advises : \mathcal{P}(FACULTY \times STUDENTS)$

Thus *advises* is a set of pairs, for example:
advises $\mathrel{\widehat=}$ {(Jones, Green), (Hartrum, Smith), (Hartrum, Adams)}

This is defined as a ***relation***, or pair-wise mapping between two sets. As a shorthand notation to the above we write

$advises : FACULTY \leftrightarrow STUDENTS$

A relation defines an m:n mapping. That is, in general, one member of the first set can participate in the relation with any number of members of the other set, and *vice-versa*. In the above example, a faculty

member may advise several students and, mathematically, a student could have more than one advisor (e.g. co-advisors). Also, in general, a relation represents a partial mapping. Not every member of either set is required to participate. Thus some faculty members may not be advising anyone, and some students may not yet have been assigned an advisor.

Since relation is itself a set of pairs, all set operators can be applied. There are several ways of expressing that a specific pair is in the relation. These are all propositions or predicates that will evaluate to TRUE if the pair is in the relation.

(Hartrum, Smith) $\in$ *advises*

Hartrum $\mapsto$ Smith $\in$ *advises*

The second form uses the *maplet* notation to represent a pair. Membership can also be defined using *infix* notation xRy.

Hartrum *advises* Smith

Alternatively the *prefix* notation $R(x, y)$ can be used.

advises(Hartrum,Smith)

Using relations, the following questions can be asked by asserting the corresponding predicate.

Is Jones an advisor?
$\exists s : STUDENTS \bullet$ (Jones, s) $\in$ *advises*

Is Green advised by anyone?
$\exists f : FACULTY \bullet$ (f,Green) $\in$ *advises*

Is Green's advisor Jones?
(Jones, Green) $\in$ *advises*

Domain and Range. Since a relation is a partial mapping, we want to be able to define the subsets of each set that *do* participate.

Given that $R : X \leftrightarrow Y$
then the *domain* (written "dom R") is the subset of X whose members participate in R, and the *range* (written "ran R") is the subset of Y whose members participate in R.

$$\mathrm{dom}\, R = \{x : X \mid (\exists y : Y \bullet (x, y) \in R)\}$$
$$\mathrm{ran}\, R = \{y : Y \mid (\exists x : X \bullet (x, y) \in R)\}$$

Or, using set comprehension,

$$\mathrm{dom}\, R = \{x : X;\ y : Y \mid (x, y) \in R \bullet x\}$$
$$\mathrm{ran}\, R = \{x : X;\ y : Y \mid (x, y) \in R \bullet y\}$$

Then in the previous example,

dom *advises* = {Jones, Hartrum}
ran *advises* = {Green, Smith, Adams}

Now we can ask if Jones is an advisor by asserting
Jones $\in$ dom *advises*

We can ask if Green is advised by anyone by asserting
Green $\in$ ran *advises*

We can ask if all students have been assigned an advisor by

$\forall s : STUDENTS \bullet s \in \text{ran}\, advises$
or
$\text{ran}\, advises = STUDENTS$

Relational Inverse If relation R is the set of pairs (x, y), then we define the relational inverse as the set of pairs (y, x). Using set comprehension we formally define the relational inverse.

$R^{-1} = \{x : X;\ y : Y \mid (x, y) \in R \bullet (y, x)\}$

For example

$advises^{-1} = \{$(Green, Jones), (Smith,Hartrum), (Adams,Hartrum)$\}$
and we can assert that Jones is an advisor by
$\text{Jones} \in \text{ran}\, advises^{-1}$

Restriction Suppose we are interested in those students advised by Hartrum or Hobart. We can write a predicate using *domain restriction.*

$\{$Hartrum, Hobart$\} \lhd advises$

This creates a subset of *advises* in which only Hartrum and Hobart are in the domain. In our example this would be {(Hartrum,Smith),(Hartrum,Adams)}. Similarly we can define *range restriction* $R \rhd T$, as well as domain and range subtraction using $\lhd$ and $\rhd$.

More formally,

$S : \mathcal{P}X$
$T : \mathcal{P}X$
$R : X \leftrightarrow Y$
$S \lhd R = \{x : X;\ y : Y \mid x \in S \land (x, y) \in R \bullet (x, y)\}$
$or S \lhd R = \{x : X;\ y : Y \mid x \in S \land (x, y) \in R\}$
$R \rhd T = \{x : X;\ y : Y \mid y \in T \land (x, y) \in R\}$
$S \lhd R = \{x : X;\ y : Y \mid x \notin S \land (x, y) \in R\}$
$R \rhd T = \{x : X;\ y : Y \mid y \notin T \land (x, y) \in R\}$

Further examples are:

$advises \rhd \{$Green,Smith$\} = \{$(Jones,Green),(Hartrum,Smith)$\}$
$\{$Hartrum, Hobart$\} \lhd advises = \{$(Jones,Green)$\}$
$advises \rhd \{$Green,Smith$\} = \{$(Hartrum,Adams)$\}$

Composition Let *COURSES* be the set of courses offered.

$COURSES \mathrel{\widehat=} \{$CSCE594, MATH531, EENG321,...$\}$
$enrolled : STUDENTS \leftrightarrow COURSES$

For example, $enrolled = \{$(Green, CSCE594), (Green, CSCE689), (Smith, CSCE594),...$\}$
We are interested in *has_students_in*, the relation relating a faculty member to courses in which his or her advised students are enrolled. We can express this using *composition*, designated with "$\fatsemi$".

$has\_students\_in = advises \fatsemi enrolled$

$R : X \leftrightarrow Y$
$S : Y \leftrightarrow Z$
$R \fatsemi S = \{x : X;\ y : Y;\ z : Z \mid (x, y) \in R \land (y, z) \in S \bullet (x, z)\}$

Relational Image Suppose we want the set of students advised by Hartrum or Hobart.

ran({Hartrum,Hobart} $\lhd$ *advises*)

As a shorthand we write $advises(\!|\{Hartrum, Hobart\}|\!)$

We define the relational image as follows.
$R(\!|S|\!) = \{x : X;\ y : Y \mid x \in S \land (x, y) \in R \bullet y\}$

A.9 Functions

As discussed earlier, a relation defines an m:n association between two sets, where membership in the relation is optional. Using *functions* allows more stringent constraints on such an association. Thus a function is a relation (i.e. an ordered pair) that is m:1. More specific constraints can be applied by using *total* and *partial* functions, as well as total and partial *injections* and *surjections*, and *bijections*. The types of functions are summarized in Table 9.

A.9.1 Total and Partial Functions

A *partial function* ($X \pfun Y$) is a relation defined on sets X and Y where not every element of X or Y needs to participate, and for each element of X that does participate there is exactly one member of Y paired with it. Note that a member of Y that participates may be related to more than one member of X.

$$X \pfun Y = \{R : X \rel Y \mid ((\forall x : X;\ y, z : Y) \bullet (x, y) \in R \land (x, z) \in R \Rightarrow y = z)\}$$

If ID is the set of all possible ID numbers, then
$id\_num : ID \pfun PERNAMES$
1. Each ID has a single name associated with it (m:1).
2. A name may have two or more IDs associated with it (m:1).
3. Some IDs may not be assigned to any name (partial).
4. Some names may not have an ID assigned (function).

A *total function* ($X \fun Y$) is a function defined on sets X and Y where *every* element of X must participate. Note that a member of Y that participates may be related to more than one member of X.

$$X \fun Y = \{f : X \pfun Y \mid \mathrm{dom}\, f = X\}$$

If $STUDENT$ is the set of all students,
and $GRAD\_CLASS$ is the set of all graduating classes, then
$is\_in : STUDENT \fun GRAD\_CLASS$ (student is member of a graduating class).
1. Each student is in a single graduating class. (m:1).
2. A graduating class has many students associated with it (m:1).
3. Every student is in some graduating class (total).
4. Some graduating classes may have no students assigned (by definition of function).

A.9.2 Total and Partial Injections

An *injection* is a function defined on sets X and Y whose inverse is also a function. In a *partial injection* ($X \pinj Y$) not every member of X need participate, while in a *total injection* ($X \inj Y$) *every* element of X must participate. Note that not all members of Y need to participate, but a member of Y that *does* participate can be related to only one member of X (1:1).

$$X \pinj Y = \{f : X \pfun Y \mid (\forall x_1, x_2 : \mathrm{dom}\, f \bullet f\ x_1 = f\ x_2 \Rightarrow x_1 = x_2)\}$$

$$X \rightarrowtail Y = \{f : X \rightarrow Y \mid (\forall x_1, x_2 : \mathrm{dom}\, f \bullet f\ x_1 = f\ x_2 \Rightarrow x_1 = x_2)\}$$

student_faculty_marriage : $STUDENT \rightarrowtail\!\!\!\!\!\!\rightarrow FACULTY$

1. Each student may be married to one faculty (1:1).
2. A faculty may be married to only one student (1:1).
3. Not every student need be married to a faculty (partial).
4. Some faculty may not be married to a student (by definition of injection).

If BOX is the set of all student mailboxes,

mailboxes : $STUDENT \rightarrowtail BOX$

1. Each student is assigned a single mailbox (1:1).
2. A mailbox has only one student assigned (1:1).
3. Every student *must* be assigned a mailbox (total).
4. Some mailboxes may not be assigned (by definition of injection).

A.9.3 Total and Partial Surjections

A *surjection* is a function defined on sets X and Y where every element of Y must participate. A member of Y may be related to more than one member of X (m:1). For a *partial surjection* ($X \twoheadrightarrow Y$) not every element of X need participate, while for a *total surjection* ($X \twoheadrightarrow Y$) *every* element of X must participate.

$$X \twoheadrightarrow Y = \{f : X \nrightarrow Y \mid \mathrm{ran}\, f = Y\}$$
$$X \twoheadrightarrow Y = \{f : X \rightarrow Y \mid \mathrm{ran}\, f = Y\}$$

If $TOPIC$ be the set of all student project topics, then

team_project : $STUDENT \twoheadrightarrow TOPIC$

1. Each topic must be selected by a student (surjection).
2. Every student has a topic (total).
3. Several students (a team) may have the same topic (*not* injective).

class_advisor : $FACULTY \nrightarrow\!\!\!\!\!\!\rightarrow GRAD\_CLASS$

1. Each graduating class has a class advisor (surjection).
2. Not every faculty member has a class to advise (partial).
3. Several faculty may co-advise the same class (*not* injective).

A.9.4 Total and Partial Bijections

A *bijection* ($X \rightarrowtail\!\!\!\!\!\!\rightarrow Y$) is a function defined on sets X and Y that is both an injection and a surjection. That is, it is 1:1 and every member of both X and Y participate.

individual_project : $STUDENT \rightarrowtail\!\!\!\!\!\!\rightarrow TOPIC$

1. Each topic must be selected by a student (surjection).
2. Every student has a topic (total).
3. Each topic is assigned to exactly one student (injection).
4. Each student has exactly one topic (function).

Note in Table 9 that a *Partial Bijection* can also be defined [1] [2]. However, no special symbol is defined. The partial bijection is a redundant function. By reversing the domain and range sets, the same relationship can be expressed as a total injection.

A.9.5 Total and Partial Finite Functions

Potter *et al* also define total and partial *finite* functions to be used in specifications where the use of infinite sets causes a problem [1]. Although they are included in Table 9 they are discussed no further in this report.

Table 9: Types of Functions

Multiplicity	Domain Membership	Range Membership	Formal Specification	Symbol	Alternate Symbol
m:n	optional	optional	Relation	$\leftrightarrow$	
n:1	optional	optional	Partial Function	$\pfun$	
n:1	required	optional	Total Function	$\fun$	
n:1	optional	required	Partial Surjection	$\psurj$	
n:1	required	required	Total Surjection	$\surj$	
1:1	optional	optional	Partial Injection	$\pinj$	$\mathbb{C}$
1:1	required	optional	Total Injection	$\inj$	$\subset$
1:1	required	required	Bijection	$\bij$	$\in$
1:1	optional	required	Partial Bijection	None	
			Finite Partial Function	$\ffun$	
			Finite Partial Injection	$\finj$	$\mathbb{C}$

Applying Functions Given:

$x : X$
$y : Y$
$f : X \fun Y$

We "apply" the function f by writing "fx" similar to the way we would write "$f(x)$" in mathematics. Note, however, that parentheses can be used anywhere to clarify syntax, the use of "$f(x)$" to represent applying function f to x is acceptable.

If we assume that each student has a single advisor we can define the following.

$advisor : STUDENTS \fun FACULTY$

Now "is Green's advisor Jones ?" can be asserted as follows.
$advisor$ Green = Jones

Similarly we can ask "Is Hobart an advisor ?" as follows.
$\exists s : STUDENTS \bullet advisor\ s =$ Hobart

Since functions are relations, which are themselves sets, all set and relation operators can be applied to functions. *However*, such operations on functions do not always produce a function.

$thesis\_advisor \mathrel{\widehat=} \{$(Green, Jones),...$\}$(every student has a unique thesis advisor).
$class\_advisor \mathrel{\widehat=} \{$(Green, Hobart),...$\}$(every student has a unique class advisor).
$advisors = thesis\_advisor \cup class\_advisor = \{$(Green, Jones), (Green, Hobart),...$\}$

The resulting *advisors* is a relation, but *not* a function.

Function Overriding *Function overriding* $(f \oplus g)$ is used to modify or update previously defined pairs of a function.

$f : X \fun Y$ existing function to be updated
$g : X \fun Y$ update function
$f \oplus g = (\mathrm{dom}(g) \domsub f) \cup g$

In other words, delete from f all pairs whose first elements match those of g, then union the result of this with g.

$advisor$ = {(Green,Jones),(Smith,Hartrum),(Adams,Hartrum)}
new = {(Smith, Hobart),(Brown,Hartrum)}
$advisor \oplus new$ =?
dom new = {Smith, Brown}
dom $new \lhd advisor$ = {(Green,Jones),(Adams,Hartrum)}
(dom $new \lhd advisor$) $\cup$ new = {(Green,Jones),(Adams,Hartrum),(Smith, Hobart),(Brown,Hartrum)}
$advisor \oplus new$ = {(Green,Jones),(Adams,Hartrum),(Smith, Hobart),(Brown,Hartrum)}

Lambda Expressions A *lambda expression* is an alternative way of writing functions. The general form is

$$\lambda\ Signature \mid Predicate \bullet Term$$

where the signature establishes the types of the variables used, the predicate gives a condition which the first element of every pair in the function must satisfy, and the term gives the form of the second element of each pair of the function.

In general, a lambda expression maps types made up from variables of the signature into the expression represented by the term for which the predicate is true. It allows you to express a function without explicitly naming the function. It can be thought of as an implicit set definition.

Examples:

$$\begin{array}{lcl} \lambda\ a,b,c : \mathcal{N} \mid a+b=c \bullet a^2+b^2+c^2 & = & \{((0,1,1),2),((2,2,4),24),\ldots\} \\ \lambda\ m : \mathcal{N} \mid m>4 \bullet m+5 & = & \{(5,10),(6,11),\ldots\} \\ \lambda\ x : 3\,..\,15 \mid x^2 \leq 9 \bullet x & = & \{(3,3)\} \end{array}$$

These are equivalent to the following set comprehension form.

$$\{a,b,c : \mathcal{N} \mid a+b=c \bullet ((a,b,c),a^2+b^2+c^2)\}$$
$$\{m : \mathcal{N} \mid m>4 \bullet (m,m+5)\}$$
$$\{x : 3\,..\,15 \mid x^2 \leq 9 \bullet (x,x)\}$$

A.10 Sequences

Sets are by definition (1) unordered; and (2) devoid of repeated elements. Often in modeling specifications an ordered sequence is needed. For example, to determine distinguished graduates, a list of students, ordered on GPA, is needed.

$ranking \mathrel{\widehat{=}} \langle$Adams, Smith, Brown, Jones$\rangle$

Here $\langle\ldots\rangle$ is used to represent an enumerated sequence.

One way to handle this problem would be to define a function as follows.

$ranking_1 \mathrel{\widehat{=}}$ {(1,Adams), (2,Smith), (3,Brown), (4,Jones)}

Using a shorthand notation, we can define *seq* as follows.

Given set X
seq $X = \{f : \mathcal{N} \pfun X \mid \mathrm{dom}\, f = 1\,..\,\#f\}$

Using this we could then declare:

$ranking$: seq $STUDENTS$

Some other examples are:

$PERNAMES$: seq $CHAR$
$SSAN$: seq $DIGITS$ where $\#SSAN = 9$

Operations Let $\text{seq}_1 X = \text{seq}\, X \setminus \langle\rangle$ which disallows the null sequence.

Consider the four operations on seq_1: *head*, *last*, *tail*, and *front*. The first two return an *element* while the last two return a *sequence*, including the null sequence.

$$head, last : \text{seq}_1 X \rightarrow X$$
$$tail, front : \text{seq}_1 X \rightarrow \text{seq}\, X$$

for $s : \text{seq}_1 X$,
$head\, s = s1$
$last\, s = s(\#s)$
$tail\, s = \{1\} \lhd s$
$front\, s = \{\#s\} \lhd s$

Recall that a sequence is a function. Hence $s\,1$ represents the application of the function to the element "1" which for a sequence returns the first element. Similarly, the domain subtractions remove the first and last elements of the sequence, respectively.

$head\ ranking$ =Adams
$last\ ranking$ =Jones
$tail\ ranking$ = ⟨Smith, Brown, Jones⟩
$front\ ranking$ = ⟨Adams, Smith, Brown⟩

Concatenation Concatenation, as its name implies, allows two sequences to be joined end to end. The infix symbol "⌢" is used.

$s^\frown t = s \cup \{n : 1 \mathinner{..} \#t \bullet ((n + \#s), t\,n)\}$
$losers \mathrel{\widehat=} \langle$Clark, Davis$\rangle$
$ranking^\frown losers = \langle$Adams, Smith, Brown, Jones, Clark, Davis$\rangle$

A.11 Tuple Concepts

Many sets are defined over several variables, making each member of the set a *tuple*. Special cases of this are those made up of 2-tuples, or ordered pairs, which form the basis for relations and functions. However, higher order tuples are frequently encountered, and have been formalized by Codd under the concept of *tuple-algebra* and *tuple-calculus* [16].

For example, consider a representation of students as follows.

$$Students \mathrel{\widehat=} \{ssan : SSAN;\ name : PERNAMES;\ age : \mathcal{N}_1 \mid \text{Person with these attributes is a student}\}$$

Thus each student is represented by a 3-tuple $(ssan, name, age)$. We would like to be able to express predicates such as "the age of all students is at least 21." In other words, we need a notation to represent the individual components of a set member. We do this through tuple notation or "dot notation."

$$\forall s : Students \bullet s.age \geq 21$$

Consider the question "what is Smith's age?"

$$\{a : \mathcal{N}_1 \mid (\exists s : SSAN \bullet (s, \text{Smith}, a) \in Students) \bullet a\}$$

Note that if names are unique, this will generate a singular set consisting of Smith's age, but if names are *not* unique the result will be a set of ages of all persons named Smith.

Consider the function *Age* written in Lambda notation as follows.

$$Age \mathrel{\widehat=} \lambda\, ssan : SSAN;\ name : PERNAMES;\ age : \mathcal{N}_1 \mid (ssan, name, age) \in Students \bullet age$$
$$Age = \{(ssan, name, age), age\}$$

Then $Age\, s$ returns student s's age. That is, s.age represents applying this function to s.

B Conference Paper Reprint

This paper was presented at the 7th SEI CSEE Conference, San Antonio, Texas, January, 1994, and is published in the proceedings in *Software Engineering Education*, Jorge L. Díaz-Herrera (ed.), Lecture Notes in Computer Science 750, Springer-Verlag, New York, 1994.

Teaching Formal Extensions of Informal-Based Object-Oriented Analysis Methodologies

Thomas C. Hartrum and Paul D. Bailor

Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45433-7765

Abstract. Teaching formal methods of software specification is often difficult. This is in part due to the lack of well defined methodologies for applying formal methods to large software system specifications. We have integrated formal specification with a more informal object-oriented modeling methodology. This allows the students to follow an established modeling approach and still generate formal specifications. We find that the students learn the formalism much easier with this approach than with our prior technique of teaching formal methods as a separate block of instruction. However, the lack of good computer-aided tools for some formal specification languages can prevent the students from directly seeing all of the benefits of using formalism. This paper describes our use of *Z schemas* to add formalism to the object-oriented modeling methodology of Rumbaugh, et.al. [RBP+91], describes the introductory software engineering course in which it is taught, and discusses our experience.

1 Introduction

Formal methods have evolved over recent years to the point of being supported by specific languages. At one end of the spectrum there exist mathematically-based, non-executable languages such as *Z* [PST91] [Inc88] [Spi89] [Hay87]. At the other end of the spectrum there exist mathematically-based, wide-spectrum languages such as Refine^{TM} [RS90] which are also executable. However, we have found the teaching of formal methods to be difficult. Part of the problem is the abstractness of the formal-based languages, which is a problem that can be easily overcome by simply doing a better job of teaching applied discrete mathematics at the undergraduate level. A more substantial problem to overcome is the lack of well defined methodologies for applying formal methods to real problems. While progress is certainly being made in this area, much still needs to be accomplished.

In our software engineering curriculum, we have developed an introductory approach that integrates well-established informal methods with formal methods. The first course in our software engineering curriculum covers object-oriented system modeling and is based on the book by Rumbaugh, et.al. [RBP+91]. We extend this approach by using the *Z* formal specification language to produce a formal-based, object-oriented specification. Our modeling language is an extension of the object-oriented model to include formal specification of its basic constructs, and our methodology builds on that already evolving for object-oriented modeling. The result is a process

that is easy to understand and apply, while resulting in a formal specification.

Given that we admit formal-based methodologies are still incomplete, a valid question to ask is "why do this?" To some extent this was addressed by the speakers at the Sixth Conference on Software Engineering Education [Sle92]. One of the more detailed presentations was that by Garlan [Gar92] which presented four course models and the advantages/disadvantages of each. We find the in-depth Master's level course to be extremely effective as well, and we would like to add a few more reasons for teaching formal methods.

1. The development of formal specifications forces students to apply the discrete mathematics learned in lower level computer science courses. This not only gives them more experience at being mathematically precise as other engineering disciplines are required to be, but it also provides an environment in which proof obligations on the specification can be conducted.

2. Formal specifications tend to be much more loosely coupled than the specifications produced by informal methods. Thus, they really do serve to specify classes of possible behaviors as opposed to very specific behavior which leads to very specific solutions.

3. Formal specifications more clearly show the benefits of reuse at higher levels of abstraction. As stated above, they more clearly specify classes of possible behaviors which means they can be more readily reused. Additionally, their mathematical basis lends them to be more easily combined by mathematical means in order to compose higher level behaviors. This capability is especially important if we are ever to realize the benefits of domain analysis and domain modeling.

4. The formality of the specification languages shows students how greatly increased automated capabilities can be obtained over current generation CASE tools. In general, this is true from both an analysis of properties perspective and a generation/synthesis of lower level code perspective.

5. Having students learn and apply these techniques does not affect their ability to use other, less formal techniques. Quite the contrary, it makes them all the more aware of their shortcomings!

Another important consideration in teaching formal methods is the question of executable versus non-executable specification languages. Both types have their merits. In our case, we chose the non-executable language *Z* for the introductory course described in this paper. Specifically, we like *Z* because it *forces* the student to think more abstractly by removing the all too familiar programming level terms from their use. When using executable, wide spectrum languages like that provided with the Software Refinery environment [RS90], the students can still rely on these lower-level concepts. However, one should not get the impression that *Z* is completely a "pencil and paper" language. We do have simple tools for syntax checking, type checking, and pretty-printing the *Z* specifications, and we have found these tools to be sufficient

for an introductory course. While more sophisticated tools for *Z* are under development, the only other tool we would like to have for such an introductory course is a general-purpose theorem prover. For our advanced course in formal methods, a more significant capability is required, and we use the Software Refinery environment and associated applications tools to treat more advanced topics like software synthesis from specifications.

The remaining sections of this paper describe our approach for integrating informal-based object-oriented analysis methodologies with formal-based specification. Specifically, Section 2 defines the informal model that is the basis for our approach, and Section 3 defines our formal extensions. Section 4 describes our work on extending the approach to also cover object-oriented design and implementation. Section 5 presents the course structure. Lastly, we conclude this paper with a discussion of the results of using this approach in four course offerings over the last two years.

2 The Informal OOA Model

The informal object oriented analysis model used for the basis of our approach is that of Rumbaugh [RBP+91]. This specific model was chosen largely because the text was already being used in our object oriented modeling course; however, the formal extensions presented here could as easily be used with the approach of Coad and Yourdan [CY91] or that of Shlaer and Mellor [SM88]. (Extensions to Booch's work have not been explicitly considered [Boo91]). This section briefly reviews the basic Rumbaugh model; for more detail, especially diagram syntax, the interested reader is referred to [RBP+91].

Rumbaugh's model consists of three parts: the *object model*, which captures the structural properties of objects and their relationships to each other; the *dynamic model*, which captures the control aspects of the object which change over time; and the *functional model*, which captures the transformation of data values within a system of objects. In each part, the model consists of a graphical representation, augmented by a *data dictionary* to provide a more detailed natural language description of the system. Our approach uses *Z schemas* to add formality to both.

2.1 The OOA Object Model

The object model captures the static structural properties of objects, and their relationships to each other. The object model is represented graphically by the *object diagram*, essentially a traditional Entity-Relationship (E-R) diagram [KS86].

Objects Each *object class* is defined by a set of *attributes*, the values of which define the state of the object. Each definition is a template that represents any actual *instances* of that class. Objects are represented on the object (E-R) diagram by a rectangle containing the object class's name.

Associations Relationships are modeled as *associations* between objects. An association, similar to an object class, represents a group of possible relationships between object instances. Most associations are binary and appear on the object diagram as a line between the two object rectangles. Associations are named, with the name written next to the line, and can have attributes themselves.

Multiplicity defines how many instances of a class are associated with a single instance of the other class, and can be 1:1, 1:n, or m:n. *Membership* of an object instance in an association may be "required," indicating that all instances of that object class must participate in an instance of the association, or "optional," indicating that the object instances may or may not participate. Such factors are represented on the object diagram by a system of special symbols at each end of the line.

There are two special types of association. *Aggregation* ("part-of") allows explicit modeling of one object that is composed of other objects. A diamond is added to the end of the line connected to the parent. *Inheritance* ("is-a") allows subclasses to be defined which inherit all of the properties of their superclass. This is represented on Rumbaugh's object diagrams by adding a triangle to the association line, with the apex pointing to the superclass.

2.2 The OOA Dynamic Model

The dynamic model captures those aspects of an object and its associations that change over time. It is graphically represented by a traditional state transition diagram.

States Rumbaugh's dynamic model represents a partitioning of an object into *states*, where each state represents unique behavior with respect to the other states. States are named, and represented on a state transition diagram by ovals. States can be hierarchically decomposed into substates to help overcome the problem of state explosion.

Events Directed arcs between the state ovals represent allowable *transitions* between states. *Events* cause the transition between states, and are written as labels on the corresponding transition arcs. An event can carry *parameters*, and can be constrained by Boolean *guard conditions*.

Since an object can only be in one state at a time, transitions out of a state must be mutually exclusive. The same event can appear on more than one transition arc, making the state entered dependent on both the event and on the state in which the event occurred.

Activities and Actions A state can be viewed as an object's response to an event. Behavior associated with a state is defined as an *activity*. An *action* is associated with a transition. Although Mealy and Moore machines are generally considered alternate modeling approaches [Dav90], both activities and actions are allowed in the same Rumbaugh dynamic model, providing a rich set of modeling possibilities.

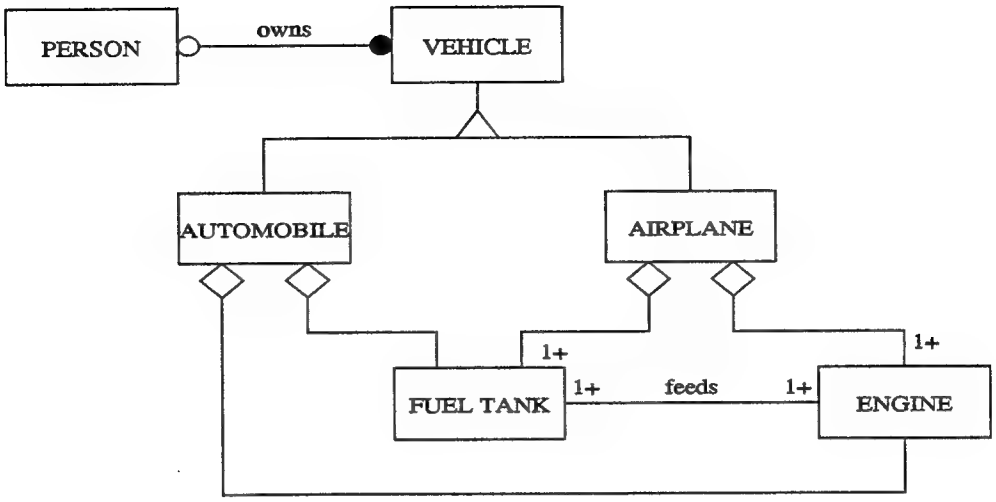

Figure 1: Informal Model Example.

2.3 The OOA Functional Model

The functional model consists of traditional data flow diagrams (DFDs) used for the classical purpose of diagramming the flow of data between processes. The functional model for an object complements the dynamic model by capturing the functional specification of the object's behavior in the sense of what calculations are to be performed, along with the sources and destinations of the input and output data of each calculation. Process descriptions are developed for the lowest level, or *leaf* processes only. Data exchanges with other objects are handled in two ways. To simply read or write another object's attribute values, that object is represented by a data store. To send or receive data as a parameter of an event, the sending or receiving object is shown as an actor (terminator). If data is to be stored or retrieved from an association, then the association is shown as a data store.

2.4 Example

Consider a system of vehicles and owners. A person can own zero or many vehicles, but each vehicle can only be owned by zero or one person. There are two subclasses of vehicle, automobiles and airplanes. An automobile has one engine and one fuel tank. An airplane has many fuel tanks and many engines, and an association between them indicating which tank feeds fuel to which engine. (Although the automobile's fuel tank also feeds its engine, the one to one mapping makes an association unnecessary). The graphical representation of the object model is shown in Figure 1.

Consider the state transition diagram for the fuel tank, shown in Figure 2. This is based on a discrete-event simulation model that includes filling the tank and using fuel from the tank. Along with the graphical representation, consider the following

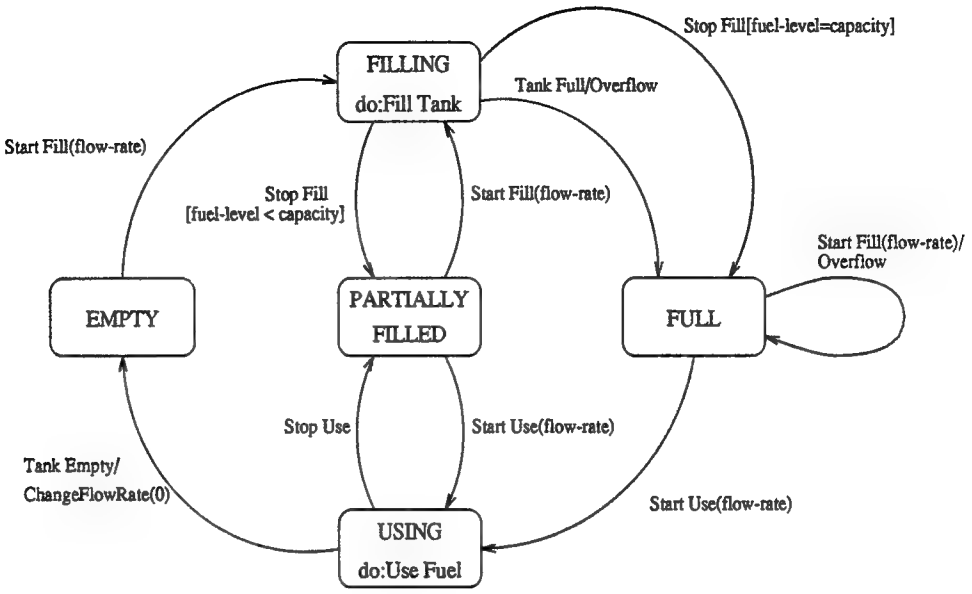

Figure 2: Fuel Tank State Transition Diagram.

three state descriptions.

Full State In this state the tank is filled to capacity with fuel. It represents a stable condition, with no input or output flow of fuel. There are no actions or activities in this state.

Filling State In this state the tank is being filled. There is no output flow of fuel. Upon entering this state the fuel tank determines when it will be full, and schedules a Tank Full event for that time. When leaving this state the fuel tank updates its fuel level. There are no other actions or activities while in this state.

Empty State In this state the tank is empty, and no fuel is flowing in or out. There are no actions or activities in this state.

Finally, consider the data flow diagrams for the fuel tank's Fill Tank activity in Figure 3. The corresponding leaf-level process descriptions are as follows.

Determine Interval Determines the (simulation) time increment between the current simulation time and the time that the attribute values were valid.

Calculate Filled Level Calculates the new level to which the tank has been filled.

Predict Overflow Time Calculates the simulation time at which the tank will overflow. This is needed to schedule the overflow event.

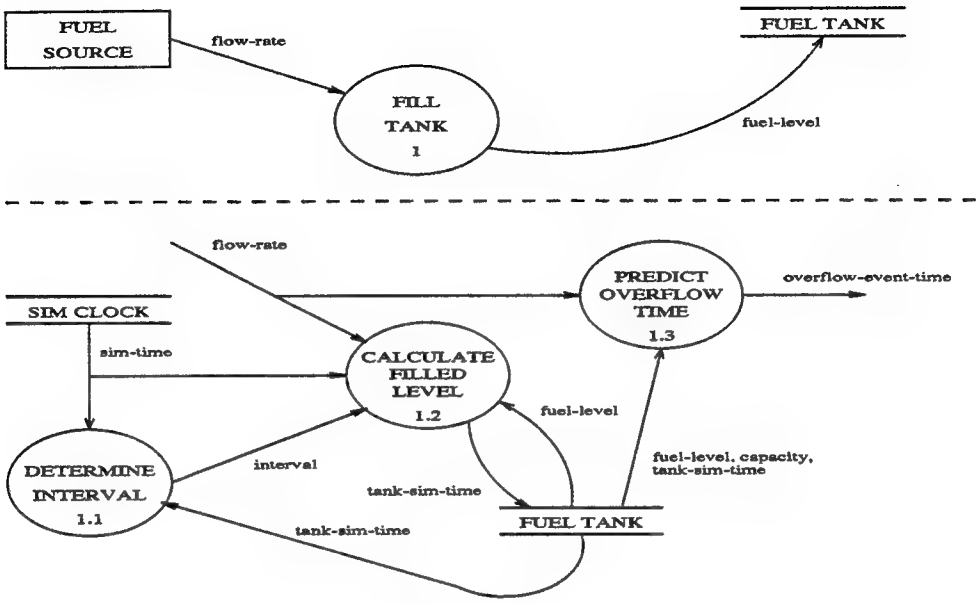

Figure 3: Fuel Tank DFD for Fill Tank.

3 Formal Extensions

Our approach is to use formal specifications to define the informal elements of the model. To do this we have chosen the language *Z* [PST91] [Inc88] [Spi89] [Hay87]. This was chosen since several references were available, it is a well-defined language that has been used in several real developments, it is in a form similar to the discrete mathematics that the students are familiar with, and unlike Refine, it does not include procedural statements, forcing the students to think in terms of the predicate calculus representation.

In *Z* both objects and operations are represented as *schemas*. As shown in Figure 4, a schema consists of a *schema name* and two parts separated by a short horizontal line. Set-theoretic variables are declared in the *signature* part which represents the union of the individual declarations. General sets are defined using all upper case letters and are not defined in any further detail. The *predicate* part is used to make statements about relationships between the variables, and while not explicitly shown, the predicate part represents the conjunction of all the statements. Figure 4 is a *static* schema, representing an entity's state space in the signature, along with invariant statements about the variables in the predicate.

A *dynamic* schema represents an operation that changes some or all of the entity's state variables. In *Z* the state variables are not traditional "programming" variables which represent storage locations that can have new values assigned to them. Rather they are *logical* variables. Thus change is represented by including both a "before" (unprimed) and an "after" (primed) version of the variables. To define the operation,

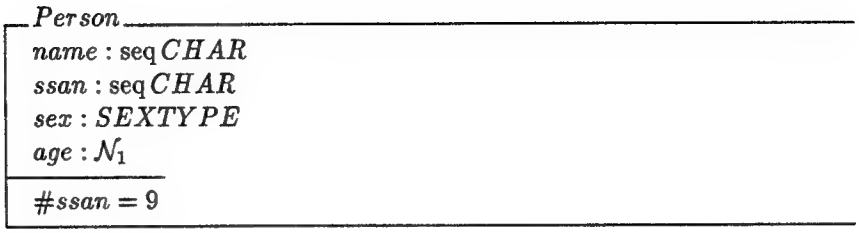

Figure 4: Static Schema for Person.

the predicate includes pre-conditions (no primes) and post-conditions (with primed variables) of the operation. Figure 5a shows an example of this. Here the *Person* static schema is being used to define a dynamic schema named *Birthday* which changes the age variable but maintains the invariant relationship on the length of the *ssan*.

Rather than explicitly including both versions in the signature part of the schema, the concept of a *Delta* schema is typically used. As shown in Figure 5b, the notation "$\Delta Person$" in the signature part of the *Birthday* schema is used to implicitly represent both the "before" (unprimed) and "after" (primed) version of the variables *and* predicates of the *Person* schema. Additionally, the signature of dynamic schemas can include input variables (decorated with a "?") and output variables (decorated with a "!").

Schemas can themselves be used as types for declaring variables. We use this fact heavily in our approach. Schemas can be combined through the use of *schema inclusion*. One schema is "included" in another by simply stating the first schema's name in the signature of the second. This has the effect of unioning both schema's signatures and conjuncting their predicates. In fact, the $\Delta Person$ schema (which itself "includes" *Person* and *Person'*) is "included in" the *Birthday* schema in Figure 5.

In our approach, we utilize static and dynamic *Z* schemas to formalize the specification of all three parts of Rumbaugh's model, the *object model*, the *dynamic model*, and the *functional model*. We describe how this is done in the following sections.

3.1 Extended Objects

The object model is formalized by defining a *Z* static schema for each object class. The signature portion of the schema defines each attribute by its name and a set-theoretic type over which it is defined. The *predicate* portion of the schema contains Boolean predicates that represent *invariants* for the object; that is, assertions that must be true at all times. Predicates can also be used to specify *derived attributes*, that is, attributes that can be derived from other attributes of the object. The *Person* schema in Figure 4 specifies the *person* object of Figure 1. Consider the model of the *Vehicle* class in Figure 6.

Birthday

$name : \text{seq}\, CHAR$
$ssan : \text{seq}\, CHAR$
$sex : SEXTYPE$
$age : \mathcal{N}_1$
$name' : \text{seq}\, CHAR$
$ssan' : \text{seq}\, CHAR$
$sex' : SEXTYPE$
$age' : \mathcal{N}_1$

$\#ssan = 9$
$\#ssan' = 9$
$age' = age + 1$

a. Expanded Dynamic Schema.

Birthday

$\Delta Person$

$age' = age + 1$

b. Abbreviated Dynamic Schema.

Figure 5: Dynamic Schema for a Person's Birthday.

Vehicle

$model\_type : MODELS$
$model\_year : YEAR$
$weight : \mathcal{N}_1$
$max\_speed : \mathcal{N}_1$

Figure 6: Static Schema for Vehicle Class.

Note that this schema has no predicate. If logically combined with another schema, the value of this predicate is taken as "true." Similarly the schemas for the fuel tank and engine objects of Figure 1 are shown in Figures 7 and 8, respectively.

FuelTank

$input\_flow\_rate : \mathcal{R}$
$output\_flow\_rate : \mathcal{R}$
$fuel\_level : \mathcal{R}$
$capacity : \mathcal{R}$

$fuel\_level \leq capacity$

Figure 7: Static Schema for Fuel Tank Object.

Engine

$model\_num : MODEL\_TYPE$
$engine\_weight : \mathcal{R}$
$engine\_fuel\_flow\_rate : \mathcal{R}$

$engine\_weight > 0$
$engine\_fuel\_flow\_rate \geq 0$

Figure 8: Static Schema for Engine Object.

3.2 Extended Associations

Each association is represented by a *Z* static schema. The set theoretic types used in the schema are the static schemas defining the object types that are related. The signature includes a form of mathematical relation (relation, function, injection, etc.) that, in most cases, captures both the multiplicity and the membership (required or optional participation) of the association.

Consider the "owns" association in Figure 1. This is 1:n and can be represented as shown in Figure 9. Here the symbol $\nrightarrow$ indicates a partial function, which captures the membership and multiplicity requirements that every owned vehicle has one owner, but not all vehicles have to be owned. The *Z* syntax includes functions to cover all other cases of multiplicity and membership. These are summarized in Table 1. The association "feeds" in Figure 1 is a special circumstance, and is explained later.

Associations can have attributes themselves. We treat all such cases as *associative objects* and define a separate schema for the associative object. Suppose in the example

Owns

$owns : Vehicle \nrightarrow Person$

Figure 9: Static Schema for "owns" Association.

Table 1: Multiplicity Representation

Multiplicity	Domain Membership	Range Membership	Formal Specification	Symbol
m:n	either	either	Relation w/ predicates	$\leftrightarrow$
n:1	optional	optional	Partial Function	$\nrightarrow$
n:1	required	optional	Total Function	$\rightarrow$
n:1	optional	required	Partial Surjection	$\twoheadrightarrow$
n:1	required	required	Total Surjection	$\twoheadrightarrow$
1:1	optional	optional	Partial Injection	$\rightarrowtail$
1:1	required	optional	Total Injection	$\rightarrowtail$
1:1	required	required	Bijection	$\rightarrowtail\!\!\!\!\twoheadrightarrow$
1:1	optional	required	Partial Bijection	None

of each vehicle being owned by a single person that such an association is characterized by a "registration number" and "registration date." These are attributes of the association, not of the person or the vehicle alone. Using Z this would be expressed as shown in Figure 10.

Registration

$reg\_number : \mathrm{seq}\, CHAR$

$reg\_date : DATE$

and

Registered

$registered : (Vehicle \nrightarrow Person) \rightarrow Registration$

Figure 10: Schemas for an Association with Attributes.

Inheritance Inheritance is a special type of association and is treated differently. We first define the superclass schema, then *include* it in the subclass schema. Consider the automobile and airplane of Figure 1. Using the *Vehicle* schema already defined, we could specify the subclasses (exclusive of their components) as shown in Figure 11. Multiple inheritance can also be specified in a similar

Automobile0

$Vehicle$

$serial\_number : \text{seq}\, CHAR$

$color : COLORS$

Airplane0

$Vehicle$

$tail\_number : \text{seq}\, CHAR$

$power : \{JET, PROP, ROTOR, GLIDER\}$

Figure 11: Schemas for Subclasses of Vehicle.

Automobile

$Vehicle$

$serial\_number : \text{seq}\, CHAR$

$color : COLORS$

$tank : FuelTank$

$engine : Engine$

$engine.engine\_fuel\_flow\_rate = tank.output\_flow\_rate$

Figure 12: Schema for Automobile Subclass of Vehicle.

manner.

Aggregation Since aggregation is a special type of association, we find it convenient to adopt a slightly different approach. The aggregate association is represented as a variable or set declared over the appropriate schema. The predicate of the aggregate object is used to "connect" the parts. Continuing the example of Figure 1, the automobile would be better represented by the schema in Figure 12.

Since an aggregate object represents an object in and of itself, we find it useful to embed associations between its component objects in the schema definition of the aggregate. Consider the "feeds" association in Figure 1. Rather than use a stand alone schema, our approach is to define it within the "airplane" schema, as shown in Figure 13.

This has several advantages. First, the "feeds" association only has meaning within the context of an airplane. Second, "feeds" can now be declared over only the fuel tanks and engines that are part of an airplane, allowing the additional predicates that specify that all tanks and all engines must participate in the association. Third, this perspective clarifies the ambiguous interpretation of Figure 1 that the associated fuel tanks and engines might be part of different airplanes.

$$
\begin{array}{|l}
\textit{Airplane} \\
\hline
Vehicle \\
tail\_number : \mathrm{seq}\, CHAR \\
power : \{JET, PROP, ROTOR, GLIDER\} \\
tanks : \mathcal{P} FuelTank \\
engines : \mathcal{P} Engine \\
feeds : tanks \leftrightarrow engines \\
\hline
power \neq GLIDER \Rightarrow \#engines > 0 \\
power \neq GLIDER \Rightarrow \#tanks > 0 \\
\mathrm{dom}\, feeds = tanks \\
\mathrm{ran}\, feeds = engines \\
\forall e : engines;\ t : tanks \bullet (t, e) \in feeds \Rightarrow \\
\qquad e.engine\_fuel\_flow\_rate = t.output\_flow\_rate \\
\hline
\end{array}
$$

Figure 13: Schema for Airplane Including "feeds."

3.3 Extended States

An object's state space is defined by: (1) the allowable values of all of its attributes, (2) its membership (or lack of) in those associations in which it can participate, and (3) the link attribute values of such associations. The object's state space can be partitioned into areas that represent unique behavior with respect to the other partitions. In the dynamic model, each of these partitions of the state space is referred to as a *state*, and is represented by a bubble on the state transition diagram. In our extended model each state is given a name, and a corresponding Z schema is defined. However, a basic static schema is inappropriate, because predicates there define invariants for all members of the class that hold for all time, while different instances of an object can be in different states. Thus for a state schema we *declare* an object of the specified type in the signature portion of the schema and *include* any involved associations. In the predicate portion, the partition of state variables that define this state is expressed as a predicate over the state variables of the declared variable using tuple notation (object.attribute) and/or membership of the input variable in the appropriate associations. These reflect invariants of that state.

A *transition* represents the change from one specific state to another. A transition is caused by an external (to the object) *event*, which may carry parameters. Parameters need to be formally declared as set-theoretic types. The transition may be conditional on some *guard condition*, a Boolean predicate expressed over the object's state variables and the event's parameter values. The transition can cause the execution of an *action*, often the sending of an event to some other object. Thus a transition can be completely characterized by an initial state, an external event, optional parameters, an optional guard condition, a target state, and an optional action.

We find the most effective way of capturing this aspect of the specification formally

is through the use of an *event transition table*, in which we group the rows by "Current State." Although a dynamic schema could be defined for each row in the table, such a presentation is not as clear due to the disjoint nature of the schemas and the amount of information that each schema must carry.

Note that if an action consists of sending an event to another object, the event name is shown in the *Action* column and must appear in the *Event* column of at least one other object's table. If the action involves more, an action name appears in the *Action* column and a corresponding dynamic schema is defined as part of the functional model.

An *automatic transition* occurs when an object has finished its specified task in a state and transitions to another state without waiting for an external event. This is shown on the state transition diagram as an unlabeled transition arrow (although it *may* have a guard condition and/or an action), and appears in the state transition table with no entry in the "Event" column. Note that this should be used to model an internal transition between two states in which the object's behavior differs, and not used to break up a state's activity into sequential steps.

As an example of using static schemas, consider the three states for the fuel tank example. The schema defining the Empty state is in Figure 14, that for the Full state is in Figure 15, and the Filling state is shown in Figure 16. The state transition table for the fuel tank is shown in Table 2.

Empty

$t : FuelTank$

$t.fuel\_level = 0$
$t.input\_flow\_rate = 0$
$t.output\_flow\_rate = 0$

Figure 14: Static Schema for *Empty* State.

Full

$t : FuelTank$

$t.fuel\_level = t.capacity$
$t.input\_flow\_rate = 0$
$t.output\_flow\_rate = 0$

Figure 15: Static Schema for *Full* State.

Filling

$t : FuelTank$

$t.fuel\_level \geq 0$
$t.fuel\_level \leq t.capacity$
$t.input\_flow\_rate > 0$
$t.output\_flow\_rate = 0$

Figure 16: Static Schema for *Filling* State.

Table 2: The Fuel Tank State Transition Table

Current State	Event	(Parameters)[Guard]	Next State	Action
Empty	StartFill	(flow_rate: $\mathcal{R}$)	Filling	
Filling	StopFill	[*fuel_level* = *capacity*]	Full	
Filling	StopFill	[*fuel_level* < *capacity*]	PartiallyFilled	
Filling	TankFull		Full	Overflow
Full	StartFill	(flow_rate: $\mathcal{R}$)	Full	Overflow
Full	StartUse	(flow_rate: $\mathcal{R}$)	Using	
Using	TankEmpty		Empty	ChangeFlowRate
Using	StopUse		PartiallyFilled	
PartiallyFilled	StartFill	(flow_rate: $\mathcal{R}$)	Filling	
PartiallyFilled	StartUse	(flow_rate: $\mathcal{R}$)	Using	

3.4 The Extended Functional Model

The extended functional model uses data flow diagrams (DFDs) in the manner specified in Section 2.3. We define a top level process bubble for each state activity or transition action. These are then decomposed as appropriate. For each leaf bubble, a *Z* dynamic schema is defined. It *includes* either the corresponding object's Δschema, in the case that the object's attribute values are modified by the operation, or the object's Ξschema, in the case that the object's attribute values are *not* modified by the operation. (Note that Ξ schemas also define both sets of unprimed and primed variables with the added constraint that their values must be equal.) Inputs that come from other active objects are defined by *Z* decorated input variables, while outputs that go to other active objects are defined by *Z* decorated output variables. Access to attributes of other passive objects should be handled by including either the Δ or Ξ schema of the corresponding object. The dynamic schemas for the DFDs of Figure 3 are in Figure 17.

DetermineInterval

$\Xi FuelTank$
$\Xi SimClock$
$interval! : SIMTIME$

$interval! = sim\_time - tank\_sim\_time$

CalcFilledLevel

$\Delta FuelTank$
$\Xi SimClock$
$input\_flow\_rate? : \mathcal{R}$
$interval? : SIMTIME$

$fuel\_level' = fuel\_level + (interval?)(input\_flow\_rate?)$
$tank\_sim\_time' = sim\_time$

PredictOverflow

$\Xi FuelTank$
$input\_flow\_rate? : \mathcal{R}$
$overflow\_event\_time! : SIMTIME$

$overflow\_event\_time! = tank\_sim\_time +$
$(capacity - fuel\_level)/input\_flow\_rate?$

Figure 17: Dynamic Schemas for Fill Tank.

4 Design Transformations

We have begun work in defining a set of transformations that can be applied to the *Z schemas* from the analysis phase that will properly map the specification into a design. Our design representation is also object-oriented, using Classic Ada^{TM} to define the appropriate objects and encapsulated methods.

The first step in our approach is to define new data types as appropriate to capture the set-theoretic types specified in the schema signatures. Then each object is transformed by mapping its static schema into a class declaration. Each signature variable becomes an attribute (instance variable), and a *set_attribute*, *get_attribute* pair of methods is defined for each. Invariants from the predicate are mapped to validation checks in the appropriate *set_attribute* methods. Associations involving attributes are handled in a similar manner, with two additional attributes that point to the associated object instances.

Methods are defined by first re-grouping the object's state transition table on

Event, then defining a method for each external event. The event parameters, if any, become input parameters for the method. Each row of the state transition table for this event type becomes a *case* in a case statement for the method. Determination of the proper case is done by conjuncting the *From State*'s schema predicate with any *Guard Condition*.

For each case three things must be done. (1) The *Current State* and *Next State* static schemas are compared, and those state variables that differ are updated. This is done by invoking the appropriate *set_attribute* methods in order to assure that any constraints are met. (2) Any *Actions*, including sending events to other objects, must be specified. For other than sending events, this is handled by a procedure call in the method, to a procedure that is developed subsequently. (3) Any *Activities* defined in the *Next State* description must be specified. This, too, is handled by a procedure call at this point. This procedure will also be called by any other events that lead to the same *Next State*.

Finally, for each procedure called as a result of the actions and activities, there should be a corresponding *Z* dynamic schema in the analysis model. From this *Z* dynamic schema, a procedure is defined, using "pseudo-code" based on Classic Ada syntax. Note that the procedure definition must be based on the pre- and post-conditions contained in the dynamic schema and these should be included as code comments (similar to the notion of annotations in Annotated Ada).

At this point, the resulting design specification can be checked for syntax and consistency errors by the compiler, and in some cases the design specification is executable. All that remains for a completed Classic Ada implementation is the final definition of the operations from the design level specification.

5 Course Structure

CSCE 594 Software Analysis and Design is a one quarter, 4 credit hour graduate level class that meets for four lectures and one problem session each week. Prerequisites consist of an introduction to discrete mathematics (note that this is sufficient for this course, but not formal methods in general), and an introduction to data structures and program design. The course examines the object-oriented paradigm and the formal specification of software. Topics include object-oriented analysis (to include context analysis, problem analysis, and specification), object-oriented design (to include architectural and detailed design), and formal specification (to include model-based specifications, set theory, and predicate calculus). Hands-on experience is emphasized through the use of homework and class projects, through the use of formal/informal specification and design languages, and through the use of computer-based software development tools, where available. Specifically, the course objectives are as follows.

- To comprehend the distinction between system and software engineering.
- To comprehend and be able to apply discrete mathematics to the formal speci-

fication of software systems.

- To be introduced to the terms *domain analysis* and *software architectures* and comprehend their role in software development.
- To comprehend and be able to apply object-oriented approaches to software analysis and design.
- To be able to determine when to choose a functional or object-oriented approach to software design.
- To comprehend the issues associated with selecting analysis and design representations and their impact on the remaining phases of software development.
- To comprehend the importance of computer-assisted tools to software product engineering in terms of the generation and evaluation of various specification and design artifacts.

The following topic areas are covered in this course. It should be noted that a specific ordering is not being implied here. Some of the areas can and should be presented in an integrated manner.

1. Introduction
   (a) Lecture Hours: 3
   (b) Learning Objective: Knowledge
   (c) Topic Components: Overview of systems modeling, Overview of domain analysis, System versus software engineering, and Overview of the object-oriented approach.
2. Object-Oriented System and Software Analysis
   (a) Lecture Hours: 19
   (b) Learning Objective: Comprehension/Application
   (c) Topic Components: Mathematics for formal specification, Z formal specification language, Information/Object Model, Dynamic Model, Functional Model, Pre/Post conditions, System Specification, Domain Analysis, Domain Modeling, and Verification/Validation of system/software specifications.
3. Formal Specification of Software
   (a) Lecture Hours: 5
   (b) Learning Objective: Comprehension/Application
   (c) Topic Components: State-based model of computation, Z formal specification language, Introduction to proof obligations and theories, and Methodology of formal specification.

4. Object-Oriented Design

   (a) Lecture Hours: 8

   (b) Learning Objective: Comprehension/Application

   (c) Topic Components: System Design, Design Foundations (Design qualities, Design assessment, Design representation, and Verification/Validation of system/software designs), Architectural design (Object design, Association design, and Software architectures and domain models), Detailed design (Design operations and methods).

5. Languages for Software Analysis and Design Representation

   (a) Lecture Hours: 2

   (b) Learning Objective: Knowledge

   (c) Components: Implications of language choice and Analysis of alternatives.

6. Classical Methods of Software Design

   (a) Lecture Hours: 3

   (b) Learning Objective: Comprehension/Application

   (c) Components: Functional decomposition, Transform analysis, and Transaction analysis.

In addition, there are a set of projects involving individual and team exercises applying modeling tools and languages appropriate to each of the topic areas. Examples of projects we have used are home heating systems, library systems, cruise control systems, elevator systems, and models of rockets.

6 Discussion

One of the things we would like to emphasize is the need to integrate informal methods with formal methods in an introductory course such as this. In the first two course offerings, we taught informal object-oriented modeling for the first seven weeks of the quarter, then introduced formal methods during the last three, loosely coupling the two using objects as the basis for the Z schemas. Students tended to treat this as a change in topic, and found the three week formal portion somewhat difficult to understand as a stand-alone topic. Also, although all had taken or (mostly) waived an undergraduate class in discrete mathematics, most lacked any practical experience at applying set theory, function definition, and predicate calculus which added to the difficulty of the three week block. By introducing the mathematics and Z formalism at the beginning of the quarter, there is more time to "relearn" the mathematics over the space of the quarter, and integrating the formalism as a way to document the informal model in the data dictionary adds to the sense of purpose of the formalism.

However, some of the students still "just don't get it." They don't see the usefulness of the formal extensions. These students express the perception that "you've added unnecessary complexity and difficulty to an existing methodology." Part of this is due to the lack of an executable formal language and a general lack of automated tools for *Z* that build on the formal specification. Much of the work in performing *Z* data and operation refinement as well as the final mapping to some implementation language must be done manually. Another problem is cultural in nature. While it is improving with each new class, many of our students have seen nothing concerning the use of formal methods other than possibly proof of correctness techniques (which provided an extremely negative impression).

Note that the course described in this paper is a required course for all our graduate computer students (approximately 50 per year). Therefore, we have exposed a substantial number of students to the use of formal methods for system development. In addition to this course, we also require all of our software engineering students (approximately 15 per year) to take an additional course entitled *Formal-Based Methods in Software Engineering.* This course uses the Software Refinery environment which contains a mathematically-based, wide-spectrum formal specification language called Refine which is also executable. Refine integrates set theory, logic, objects, a formal object base, transformation rules, and pattern matching [RS90]. Additionally, the Software Refinery environment includes a parser generator and X11 interface toolkit. Essentially, Software Refinery can be used to build program transformation systems [LM91] in which high level formal specifications are used as the basis for synthesizing lower level implementations in languages like C and Ada. With the basis provided by our introductory course CSCE 594, the software engineering students are now in a position to build object model transformation systems. In particular, it is shown how object-based domain models can be used to develop domain-specific languages and how these languages can be transformed in a behavior-preserving way using object model to object model transformations. Thus, the elements of how to build a specification to code level transformation system are not only discussed in class, but practical experience with constructing and using such systems is obtained.

Overall, we feel this course has been very successful at simultaneously introducing students to formal methods and object-oriented modeling. While some problems still remain in terms of computer-aided tools for supporting the *Z* formal specification language, we feel the overall benefits greatly overcome this specific deficiency. Additionally, this course coupled with our advanced course in formal-based methods, where we do have more sophisticated tool support, produces a very enlightened set of software engineers.

References

[Boo91] Grady Booch. *Object Oriented Design with Applications.* Benjamin/Cummings Publishing Company Inc., Redwood City, CA, 1991.

[CY91] Peter Coad and Edward Yourdon. *Object-Oriented Analysis, 2nd Ed.* Yourdon Press, Englewood Cliffs, NJ, 1991.

[Dav90] Alan M. Davis. *Software Requirements, Analysis and Specification.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1990.

[Gar92] David Garlan. Formal Methods for Software Engineers: Tradeoffs in Curriculum Design. In *Proceedings of the Sixth SEI Conference on Software Engineering Education*, pages 131–140, San Diego, CA, Oct 1992.

[Hay87] Ian Hayes. *Specification Case Studies.* Prentice Hall International (UK) Ltd, Hertfordshire, 1987.

[Inc88] D. C. Ince. *An Introduction to Discrete Mathematics and Formal System Specification.* Oxford University Press, New York, 1988.

[KS86] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill, New York, 1986.

[LM91] Michael R. Lowry and Robert D. McCartney. *Automating Software Design.* MIT and AAAI Press, Menlo Park, California, 1991.

[PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall, New York, 1991.

[RBP+91] James Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[RS90] Inc. Reasoning Systems. *Refine User's Guide.* 3260 Hillview Avenue, Palo Alto, CA 94304, 1990.

[Sle92] Carol Sledge, editor. *Proceedings of the Sixth Software Engineering Education Conference.* Springer-Verlag, New York, New York, 1992.

[SM88] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis.* Yourdon Press, Englewood Cliffs, NJ, 1988.

[Spi89] J. M. Spivey. *The Z Notation, A Reference Manual.* Prentice Hall International (UK) Ltd, Hertfordshire, 1989.

References

[1] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z.* New York: Prentice Hall, 1991.

[2] D. C. Ince, *An Introduction to Discrete Mathematics and Formal System Specification.* New York: Oxford University Press, 1988.

[3] J. M. Spivey, *The Z Notation, A Reference Manual.* Hertfordshire: Prentice Hall International (UK) Ltd, 1989.

[4] I. Hayes, *Specification Case Studies.* Hertfordshire: Prentice Hall International (UK) Ltd, 1987.

[5] I. Sommerville, *Software Engineering, 3rd ed.* Wokingham, England: Addison-Wesley, 1989.

[6] I. Reasoning Systems, *Refine User's Guide.* 3260 Hillview Avenue, Palo Alto, CA 94304, 1990.

[7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1991.

[8] S. Stepney, R. Barden, and D. Cooper, *Object Orientation in Z.* New York, New York: Springer-Verlag, 1992.

[9] P. Coad and E. Yourdon, *Object-Oriented Analysis, 2nd Ed.* Englewood Cliffs, NJ: Yourdon Press, 1991.

[10] S. Shlaer and S. J. Mellor, *Object-Oriented Systems Analysis.* Englewood Cliffs, NJ: Yourdon Press, 1988.

[11] E. Yourdon, *Modern Structured Analysis.* Englewood Cliffs, NJ: Yourdon Press, 1989.

[12] F. Hayes and D. Coleman, "Coherent models for object-oriented analysis," in *Proceedings of OOPSLA '91*, (Phoenix, AZ), pp. 171–183, Oct 1991.

[13] K. J. Lee, M. S. Rissman, R. D'Ippolito, C. Plinta, and R. V. Scoy, "An ood paradigm for flight simulators, 2nd ed," Tech Report CMU/SEI-88-TR-30, Software Engineering Institute, Pittsburg, PA, Sep 1988.

[14] K. J. Lee, R. D'Ippolito, C. Plinta, and J. Stewart, "Model-based software development (draft)," Special Report CMU/SEI-92-SR-00, Software Engineering Institute, Pittsburg, PA, Dec 1991.

[15] M. R. Lowry and R. D. McCartney, *Automating Software Design.* Menlo Park, California: MIT and AAAI Press, 1991.

[16] E. F. Codd, *Relational Completeness of Data Base Sublanguages.* Englewood Cliffs, NJ: Prentice Hall, 1972.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	7 October 1994	Technical Report

4. TITLE AND SUBTITLE
A Formal Extension to Object Oriented Analysis Using Z

5. FUNDING NUMBERS

6. AUTHOR(S)
Thomas C. Hartrum
Paul Bailor, Maj, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/EN/TR-94-07

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFIT/ENG
Wright-Patterson AFB OH, 45433-6583

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes extending an informal object oriented analysis model with formal constructs. In particular, the object modeling technique (OMT) of Rumbaugh et. al. is integrated with the formal specification language "Z". The result is a software analysis process that is easy to understand and apply, while resulting in a formal specification.

14. SUBJECT TERMS
Software Engineering, Formal Specification, Object-Oriented Modeling

15. NUMBER OF PAGES
94

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to *stay within the lines* to meet *optical scanning requirements*.

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	PR	- Project
G	- Grant	TA	- Task
PE	- Program Element	WU	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD	-	See DoDD 5230.24, "Distribution Statements on Technical Documents."
DOE	-	See authorities.
NASA	-	See Handbook NHB 2200.2.
NTIS	-	Leave blank.

Block 12b. Distribution Code.

DOD	-	Leave blank.
DOE	-	Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
NASA	-	Leave blank.
NTIS	-	Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.